1984

The design, implementation and application of a table-driven, syntax-directed editor.

Tilley, George M. Jr

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE DESIGN, IMPLEMENTATION AND APPLICATION
OF A
TABLE-DRIVEN SYNTAX-DIRECTED EDITOR

by

George M. Tilley, Jr.

December 1984

Thesis Advisor: Bruce J. MacLennan

Approved for public release; distribution is unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): The Design, Implementation and Application of a Table-Driven, Syntax-Directed Editor	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis December 1984
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): George M. Tilley, Jr.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93943	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93943	12. REPORT DATE: December 1984
	13. NUMBER OF PAGES: 130
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

syntax-directed editor, context-free grammar, syntax-directed translation scheme, programming environment

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A syntax-directed editor facilitates the creation of programs in a particular programming language. Because it is based on the syntax of the language, the editor ensures the syntactic correctness of edited programs. This paper discusses the writer's development of a table-driven syntax-directed editor capable of editing information structured under virtually any context-free grammar. Not only does this editor ensure syntactically correct programs, but it also possesses (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued)

limited translation capabilities, both between high-level languages and from a high-level language into a directly executable form. The broader implications of such an editor, and of syntax-directed editing in general, are also discussed.

Approved for public release; distribution is unlimited.

The Design, Implementation and Application of a Table-Driven Syntax-Directed Editor

by

George M. Tilley, Jr.
Captain, United States Army
B.S., United States Military Academy, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

ABSTRACT

A syntax-directed editor facilitates the creation of programs in a particular programming language. Because it is based on the syntax of the language, the editor insures the syntactic correctness of edited programs. This paper discusses the writer's development of a table-driven syntax-directed editor capable of editing information structured under virtually any context-free grammar. Not only does this editor insure syntactically correct programs, but it also possesses limited translation capabilities, both between high-level languages and from a high-level language into a directly executable form. The broader implications of such an editor, and of syntax-directed editing in general, are also discussed.

TABLE OF CONTENTS

LIST OF FIGURES

I. SYNTAX-DIRECTED EDITING IN THE MODERN PROGRAMMING ENVIRONMENT

A. PROGRAMMING ENVIRONMENTS

Over the past ten years or so, computer scientists have been devoting increasing attention to the notion of a "programming environment": the set of software and hardware tools available to aid the programmer in the performance of his task. In the past, the programming environment merely consisted of disjoint systems programs that the programmer had to invoke deliberately and sequentially to input, translate, and execute his programs. Today, however, environments are designed so that individual tools are both more useful and well-integrated as parts of a whole, with the overall result that program development is facilitated rather than hindered.

As late as the 1970's, program development was an iterative and tedious process. Some of the tools in a typical programming environment were:

keypunch machine: used to enter program instructions on (usually) 80-column data cards;

card reader: a machine used to read the deck of data cards into the computer's memory;

compiler: a program that translated high-level language programs into assembler language or internal machine language. Note that if translated into assembler language, an assembler program was also required for conversion into machine language -- in fact, this program sometimes had to be provided by the programmer as part of his card deck;

linkage editor: a program that linked object programs (the machine code produced by the assembler or compiler) and certain control information before loading;

loader: a program that loaded object modules and needed library routines for subsequent execution;

line printer: a machine which usually produced the only visual output from the system described above.

The tools described above formed a strictly "batch" environment. This environment was not significantly improved with the addition of time-sharing, which basically involved the combination of input and output devices into a teletype-style terminal. However, time-sharing did give rise to stored files of programs and data, primitive editing features to create these files, and new control words (such as "RUN") to combine several compilation-to-execution primitives.

Consider the process the programmer had to follow to develop a correct program using the tools described above. After designing an algorithm to solve his real-world problem and selecting a programming language, he usually drafted the program on paper and desk-checked its correctness by stepping through the program one statement at a time. When he was satisfied that his program was correct, he keypunched it onto data cards and combined them with the necessary control cards to invoke the tools he desired. A typical card deck included such cards as:

job card: to uniquely identify the program while in the computer;

compiler card: to invoke the compiler of the chosen language;

the program cards;

assembler card: to invoke the assembler (required if the compiler's output was assembly language and not machine code);

the assembler program: if required, as a deck of cards;

load card: to invoke the (usually system-provided) loader;

object modules: program portions previously compiled for inclusion in this program (reusable subroutines, for example);

data card: to tell the system that input data followed;

input data cards;

end card: to signify the end of the data (and the job) [Ref. 1: p. 200].

After preparing the card deck, the programmer fed the deck into the card reader and waited for his output, which was usually produced on the line printer. If the program contained a compilation error, the programmer had to determine the cause of error (usually with the help of a diagnostic message of questionable utility), edit his program by typing new cards to replace the erroneous ones, and resubmit his deck through the card reader. Even if the program compiled successfully, it might have been aborted during execution because of a run-time error, again necessitating the error detection and correction procedures previously mentioned. A third error type that occurred was the logic error that compiled and executed but produced incorrect output. After checking the output and determining it was incorrect, the programmer again had to determine the cause of error (but this time without any diagnostic aid from the system) and repair the program.

Even if the expense of computer time were not a factor (which it was in that era), one can easily understand the other reason for program drafting and desk-checking: to lessen the personal frustration of program correction and resubmission [Ref. 2: p. 445]. Clearly the programming environment was not conducive to program development. It forced the programmer to concern himself with satisfying its requirements, avoiding aborted runs, and minimizing the number of job submissions, while his prime concern should have been the problem he was originally trying to solve.

It is certainly true that the hardware technology of the time played a role in causing the unfriendliness of the programming environment. However, the arrival of faster, cheaper computers and interactive time-sharing with intelligent terminals in the 1970's did little more than replace the above process with the tedious cycle of invoking an editor, editing a program, saving the program, exiting the editor, invoking the compiler, debugging, and re-invoking the editor to effect changes. Only recently has attention been directed toward taking better advantage of modern capabilities to provide a useful, productive programming environment.

Because programming environments are such a current research topic in computer science, there are many opinions as to what a modern environment should do for the user. It is safe to say, however, that it should do much more than simply correct the obvious deficiencies of previous systems as discussed above. Sandewall [Ref. 3: pp. 35-36], for example, presented a list of some of the desirable functions of a programming environment, which included administration of program modules, test cases, and documentation; interdialect translation; compatibility checking between program segments; support for a particular development methodology (such as top-down design); enhanced support of the

interactive session (to enable, for example, the programmer to back up through his commands and undo their effects); and specialized editing based on the editor's knowledge of the syntax of the programming language. Winograd [Ref. 4: p. 14] envisioned a futuristic environment as a "moderately stupid assistant, to whom we give all the information we possibly can, and who in turn relieves us of much of the burden of memory, tedious checking, and drawing more-or-less straightforward conclusions." Based on the above, it is appropriate to summarize simply that a programming environment should do everything possible to facilitate program development.

There are many "state-of-the-art" programming environments in operation, each possessing somewhat different capabilities. Interlisp [Ref. 5], for example, provides an environment for the development of LISP programs. During interactive sessions, the user talks exclusively to the Interlisp system. The program being developed is created, stored, and manipulated as a data structure by the system's structure editor, which displays the program in textual form for the user. A facility called "Masterscope" analyzes and cross-references the program to provide such information as which functions call which, how and where variables are bound, and so on. Interlisp also includes a DWIM ("Do What I Mean") facility which, upon error detection, attempts to determine what the user intended and automatically make the necessary correction.

Another example of a modern programming environment is the Cornell Programming Synthesizer for PL/CS, a subset of the PL/I language [Ref. 6]. It allows creation and editing of programs through a syntax-directed editor, which uses templates based on the language's grammar to insure the syntactic correctness of the program. Like Interlisp, it stores the program internally in a tree structure but

displays it in textual form. The Synthesizer also includes sophisticated debugging aids that permit tracing the flow of execution through the program at any user-selected rate. The user can step the program one statement or construct at a time, and may command the system to display the value of particular variables as he does so. (This is an excellent example of an environment freeing the user from a tedious activity. Contrast such a feature with the outdated advice given in [Ref. 2: p. 453], which states that to trace a program's progress, "the use of additional WRITE commands in strategic places is the most useful technique.")

One characteristic of both environments described above, and of modern environments is general, is the integration of individual tools. The progress of a particular tool is shared with the others, with the result that the system both eliminates duplication of effort and gains knowledge about the program being developed. For example, the syntax-directed editor of the Cornell Program Synthesizer produces an executable derivation tree form of the program during the editing session; using such a structure as an interface between tools, subsequent compilation or direct execution can begin without the re-parsing which would have been required had a conventional text editor been used. The Interlisp tools are often invoked from within each other by the user, allowing him to consider program segments from different perspectives without losing his place in the program. Sharing of program knowledge among the tools thus can provide a more responsive overall environment. In the future, programming environments may even resemble Winograd's System A [Ref. 4], which comes to "understand" a program as it is being developed, forming its own comments and checking the program against the user's apparent intentions.

B. SYNTAX-DIRECTED EDITORS

A syntax-directed editor, as its name suggests, is a tool used to edit programs based on the syntax of an underlying programming language. Typically, it utilizes templates of language constructs inside of which the programmer enters such items as variable names, procedure calls, output strings, and so on. The goal of such an editor is to free the programmer from concern over syntactic issues. A program constructed with a syntax-directed editor is assured to be free of syntax errors.

One advantage of a syntax-directed editor is that overall development time may be reduced by avoiding edit sessions whose sole purpose is to correct syntax errors for subsequent re-compilation. If the editor utilizes templates, two additional advantages may be realized. First, the user need not even learn the details of the language's syntax -- he merely has to know which template to install at a given point in the program. Second, selection of templates with single keystrokes may reduce the time spent in the editing process itself when compared to typing the symbols in the constructs individually using a text editor.

Syntax-directed editors typically represent the program they are editing as a tree structure, and present a textual image to the user through the templates he has selected. If the internal representation of the program is in fact a structure suitable for subsequent interpretation or code generation, then the editor serves as a parser as well. In terms of the overall environment, this allows the presence of a much simpler (and faster) compiler or interpreter needing no scanner, parser, or syntax error recovery mechanism. Some limited facility for program translation from one high-level language to another may also be possible, if

such translation amounts to direct substitution of one set of templates for another.

There are several examples of syntax-directed editors in widespread use today. One that most nearly matches the description above is that found in the Cornell Program Synthesizer [Ref. 6]. It uses templates for PL/CS grammatical constructs and creates programs top-down by inserting templates and phrases into the existing templates. However, this editor is more than a simple syntax-directed editor because it insures a degree of semantic correctness as well. For example, it identifies variables that are referenced but not declared. As part of the overall programming environment, its product is directly executable by other tools. Even when a program has not been fully created, it can be interpreted up to the point of incompletion. New code can then be entered, and interpretation can resume.

Interlisp [Ref. 5], also mentioned above, has an editor that manipulates a program through its syntactic structure rather than its textual form. Due to the syntactic simplicity of LISP, however, this editor does not use templates; virtually any combination of atoms and lists comprise a syntactically (if not sementically) correct LISP program. Originally, Interlisp's editor was designed for teletype-style interaction and had no full-screen capability. More recently, a display-oriented editor, DED, has been included to enhance Interlisp's interactive nature [Ref. 7].

One variation on syntax-directed editing is to combine the qualities of syntax-directed editing with text editing. [Ref. 8] describes a family of such editors produced from a Hybrid Editor Generator, which receives as input a specification for a grammar and outputs an editor for that language. These Automatically Generated Editors allow the user to enter menu selections to create program segments and

navigate within a tree, as in a syntax-directed editor, but they also allow the user to enter text at any stage in the editing process. The text is parsed by the editor, and the tree produced by the parse is grafted onto the existing program tree. Another editor in this category is the "Z" editor at Yale University [Ref. 9]. It possesses syntax-directed features such as automatic indentation, automatic balancing of expressions, user-directed selection of entire syntactic units, and an adjustable level of display detail. However, since it uses a text-oriented model of a program rather than a tree structure, it is more accurate to state that Z is a text editor capable of simulating many of the functions found in a syntax-directed editor.

Whereas Z is a text editor that simulates a syntax-directed editor, there also exist syntax-directed editors that manipulate text. ED3 [Ref. 10], for example, is an editor "primarily designed for manipulation of hierarchically structured texts." It does so by superimposing a tree structure onto the text, analogous to a structured outline or table of contents. The section to be edited or viewed is selected by navigating around the tree. Further discussion of structured "document editors" may be found in [Ref. 11] and [Ref. 12].

The MENTOR system [Ref. 13] includes an editor that can accurately be called syntax-directed because it edits programs by manipulating abstract syntax trees based on the grammars of programming languages. The system utilizes a tree manipulation language, MENTOL, which includes primitives from which macros may be created to tailor the system to edit programs in a particular programming language. A viable set of Pascal macros currently exists. Note, however, that because the MENTOR system may be configured to handle any of a variety of languages, it is accurately described as "a processor designed to manipulate structured

data" [Ref. 13: p. 129] in that it can edit any information that can be structured under a format acceptable for input. The notion that syntax-directed editing may be applied to structures other than program trees is further discussed in [Ref. 14].

C. INTRODUCTION TO THE SDE AND OVERVIEW OF THIS PAPER

With the above discussion of modern programming environments and syntax-directed editing as background, this paper will discuss the writer's development of a table-driven syntax-directed editor capable of manipulating information structured under virtually any context-free grammar. This editor, hereafter called the SDE, stores, retrieves, and edits tree structures based on the rules presented in an input grammar selected by the user. Interactive in nature, it is menu-driven and terminal-independent. As will be seen, its manner of tree manipulation also gives it limited language translation and other desirable properties.

The SDE was based primarily on the work found in [Ref. 15]. It was programmed in Pascal as compiled by the Berkeley compiler, and is currently in operation within a Unix environment on a VAX 11/780 minicomputer. The reader is also invited to read [Ref. 16], which presents an in-depth discussion of table-driven syntax-directed editing and which served as research material both for this paper and for [Ref. 15].

Chapters Two, Three, and Four of this paper may be viewed as describing the SDE in progressively greater levels of detail. Chapter Two describes a sample session using the SDE and serves as an introduction to its operation. Chapter Three discusses the conceptual basis of the SDE, including the algorithms it uses to display and store information. Finally, Chapter Four discusses detailed implementation of

the SDE to include data structures used, display implementation, and data storage.

Chapter Five assesses the accomplishments of the SDE both in theory and as a product. It describes the SDE's design decisions and may serve as an "after-action report" on the SDE. Improvements and future development are also discussed. Chapter Five further contrasts syntax-directed editors with text editors and discusses the implications of syntax-directed editing in general.

II. A SAMPLE EDITING SESSION WITH THE SDE

A. GENERAL

The SDE was designed for interactive sessions using a computer terminal. It displays edited programs in textual form on the screen, but creates, manipulates, and stores them as program trees. The SDE does not hide this representation from the user. Rather, many of its commands are worded to guide the user through the tree he is editing, constantly reminding him that his real product is something other than its textual representation.

The user interacts with the editor by typing the desired command, followed by a carriage return. He can also enter a series of commands (up to 80 characters in total length) followed by a single carriage return. Illegal commands are detected and reported when entered individually; when entered as part of a string of commands, they are reported and the rest of the string is ignored.

(In the paragraphs that follow, terms such as "control-A" or "^A" refer to the consecutive striking of the "control" and "a" keys on the keyboard.)

B. INITIALIZING THE SDE

Every session with the SDE commences with the SDE presenting a series of questions to the user as follows:

GRAMMAR FILE: (Enter the name of the file containing the grammar the SDE will use to parse and display the program.)

PROGRAM FILE: (Enter the name of the file to be edited. If editing an already-existing file, the file must be in

readable format to the SDE -- which is assured if it was written by the SDE using the same grammar file in the last editing session.)

FILE ALREADY EXISTS (Y OR N)? (This question must be answered to prevent the SDE from attempting to access a file that doesn't exist. It is a limitation of the Pascal implementation of the SDE.)

The present implementation allows all of the above information to be provided on the "command line" that invokes the SDE. Thus the same initialization could be achieved by typing SDE PASCAL DEMO.P Y, for example.

(A third file, called "TERM", is also accessed by the SDE and must be present in the environment. This file provides information to the SDE about the display screen being used. Details about this file are provided in Appendix F.)

At this point in the session, the SDE has read the grammar and "TERM" files and has organized the information contained in them. If a pre-existent program file was indicated, this file has also been read and processed; the program as last edited appears on the screen. If creating a new program, a skeleton of a program (based on the selected grammar) appears. In either case, a menu of choices also appears at the bottom of the screen. The initial display of a new program using the "Minigcl" grammar in Appendix E is shown in Figure 2.1.

Note in the figure that the current focus of attention (hereafter called the "current position" or CP) is indicated by underlining. On an actual display screen, the current position is indicated by inverse video (if the terminal supports it) or by any distinguishing characters indicated in the "TERM" file.

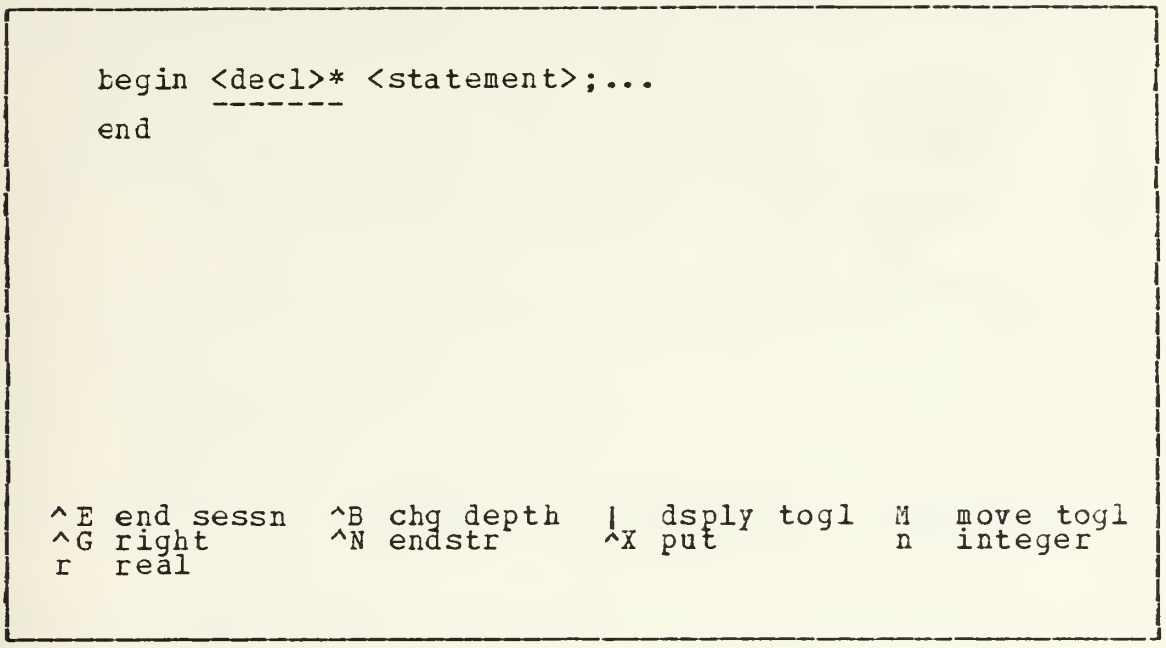

Figure 2.1 Initial Display for New Minigol Program

C. MOVING AROUND IN A PROGRAM

The following paragraphs assume the current state of editing to be as depicted in Figure 2.2. (The grammar in use is, again, the "Minigol" grammar of Appendix E.)

Figure 2.2 indicates that the Current Position is the "i < 10" portion of the "while" statement. Observe the commands the editor makes available to the user when at this CP. Setting aside the more general commands for now, one notes that the user may move the CP either to the right, to a "child," or to a "parent." These movements make sense when one realizes that he is moving through the program tree and not directly through the text. Thus, moving to the "parent" shifts the CP to the node in the program tree whose sons include the "i < 10" portion of the program; moving to the right shifts the CP to its brother node to the immediate right in the tree; and moving to the "child" shifts the CP to the leftmost son of the (present) CP.

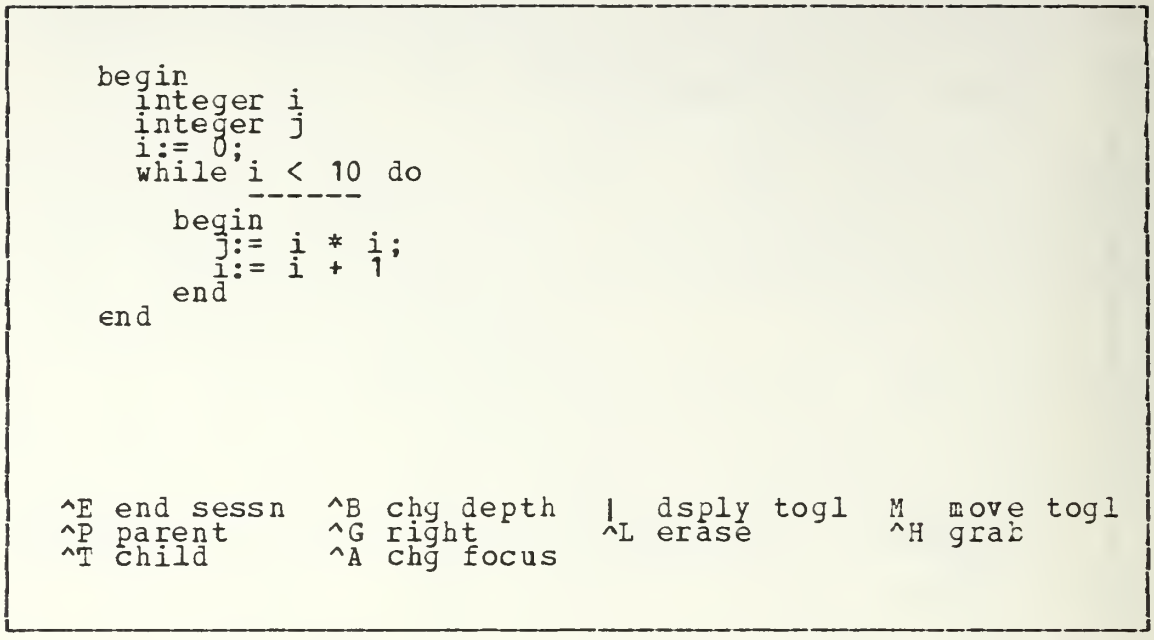

Figure 2.2 Sample State of Editing Using Minigol Grammar

(Actually, this is not entirely true. When the user selects a command to move the CP, the actual direction of movement is hidden from him. However, the apparent movement, as seen on the display, is in the direction selected by the user. For a more thorough explanation of this operation, see Section 3.C.)

As the user moves the CP about the program tree, the set of legal commands changes. For example, the command to move to a right brother is offered only if that brother exists. The entire set of movement commands (each being offered when applicable) includes "parent" (to move upward in the tree), "child" (to move downward in the tree), and "right" and "left" (to move to brother nodes on the same level in the tree). An additional command, "rest seq," applies only when the CP points to an element of a sequence, such as a sequence of individual declarations within a block. This command positions the CP to reference all subsequent

elements of the sequence, and is useful when attempting to delete or move the remaining elements.

Returning to the situation depicted in Figure 2.2, suppose the user types "control-G" to move the CP to the right. When the screen is updated, it will appear as shown

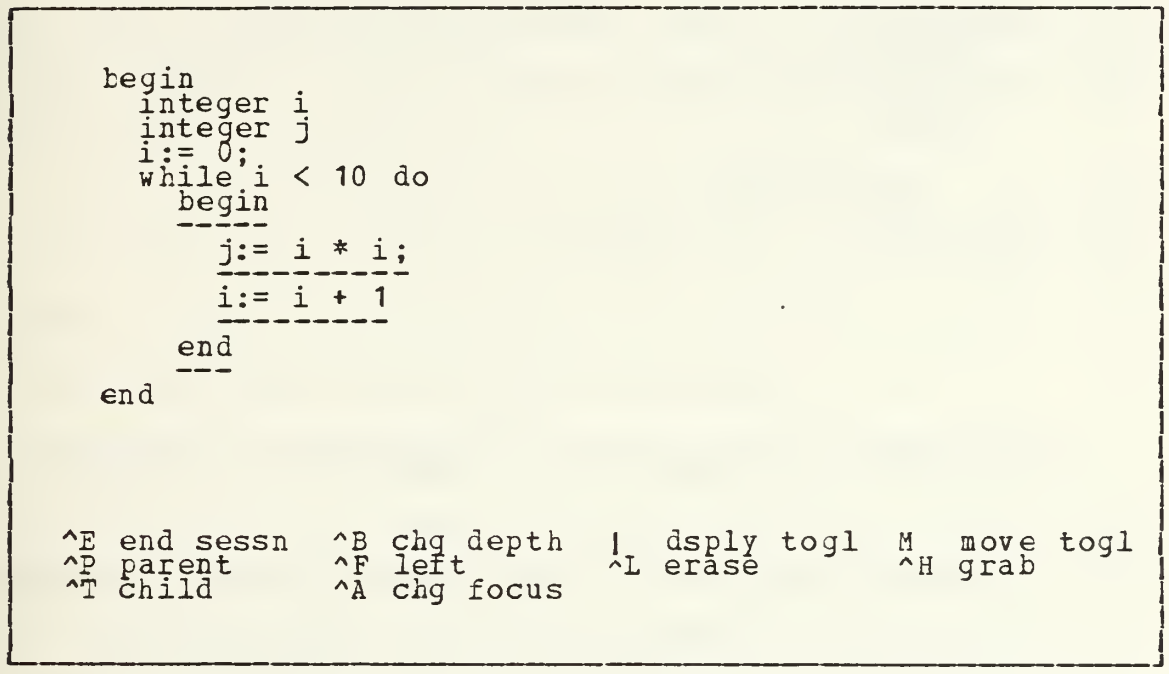

Figure 2.3 Editing State, CP = "begin-end" Block

in Figure 2.3. The menu portion is unchanged except that the command to move to the right has been removed (indicating there are no more brothers to the right) and the command to move to the left has been added (indicating the previous location of the CP). Typing "control-T" at this point causes the strange display indicated in Figure 2.4. The CP is located at a node in the program tree that has no textual representation on the screen. In this particular instance, the CP is the "declarations" portion of the "begin-end" block, which the user chose to close off in a previous editing session. A "nil" node remains in the tree,

however, and can be accessed in the same manner as any other node. Should the user later wish to insert declarations in this block, he can erase the "nil" node (using control-L), at which time the SDE will prompt him for declarations.

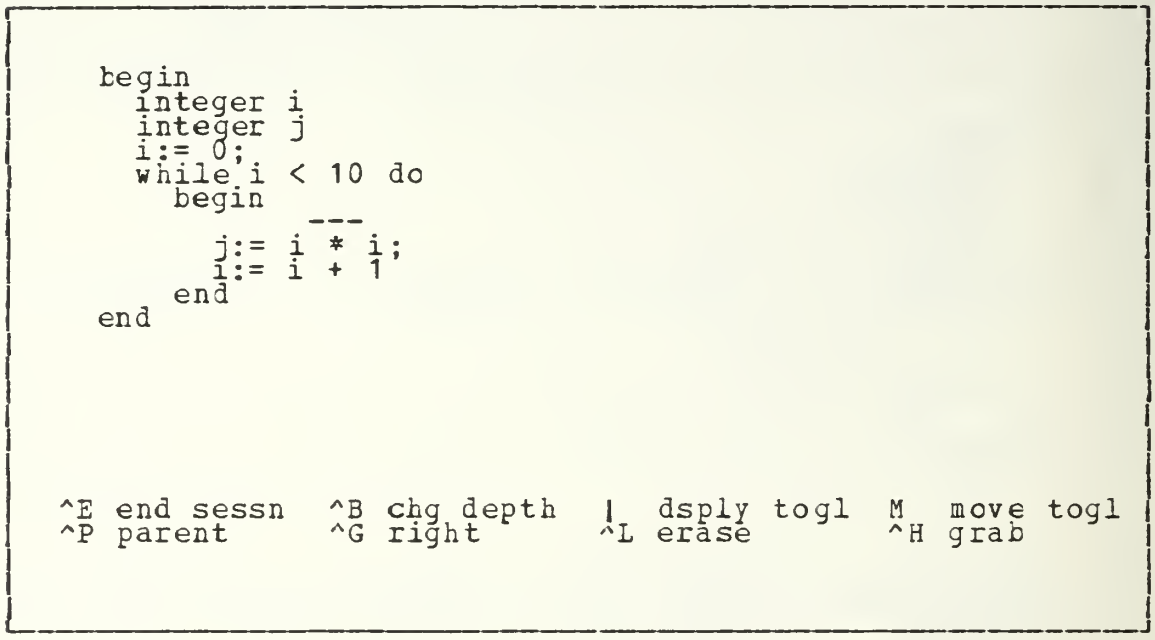

Figure 2.4 Editing State, CP = Declarations

D. EDITING A PROGRAM

Moving around in a program tree may be considered a "passive" activity in that it has no effect on the tree itself. The following paragraphs discuss those editing commands which change the program tree -- in other words, the actual "editing" functions of the SDE.

The SDE is capable of performing the following editing functions:

delete a portion of the program tree;

create a portion of the tree (by building from a nonterminal leaf in an incomplete program tree);

move a portion of a program tree from one location to another;

insert a subtree into a sequence of like subtrees.

The above capabilities describe what is performed on the tree. Viewed in terms of their textual effects, these functions enable the user to:

delete a user-defined name, a statement segment, an entire statement, or a block of statements in a single command;

add to the current program;

move text from one location tc another;

insert an item into a sequence of like items.

Note that on a more general level, the SDE allows the user to ADD or DELETE information. It does not, however, allow the user to directly CHANGE information (for example, through a "global replace" operation), although this function may be realized indirectly through a series of DELETE and ADD commands. (Reasons for this limitation of the SDE and possible corrective implementations are discussed in Chapter 5.)

Deletion of a program segment is simple. The user moves the CP until it references the entire portion to be deleted (as indicated by highlighting with inverse video on some terminals), then enters the control-L command ("erase"). That portion of the text is removed and replaced with the name of the nonterminal node type expected there. ("<decl>*" is an example of such a node type. Its presence tells the user that this portion of the program is incomplete.)

Adding to the existing tree is dependent on one's location within the tree. First, such an operation is legal only when the CP references a nonterminal node as described in the previous paragraph. When the CP references a nonterminal on the screen, it is referencing a leaf on the (incomplete) program tree to which sons must be added to complete the tree. Second, the nature of this specific nonterminal dictates the choice of possible sons from which the user may select. Referring again to Figure 2.1, the menu includes commands to select an "integer" or "real" declaration. These options would not have been offered if the CP were referencing the "statement" nonterminal instead of the "decl" nonterminal.

Assuming the user selected command "n" (for "integer") in Figure 2.1, the display would then resemble that of

```
begin
  integer <id> <decl>* <statement>;...
end




^E end sessn  ^B chg depth  |  dsply togl  M  move togl
^P parent     ^F left       ^X put          any char
```

Figure 2.5 Program Display While Creating Declarations

Figure 2.5. Note that the word "integer" has been added to the display, the CP references a new "var" nonterminal, and the menu choices reflect the new CP. The SDE's automatic

movement of the CP is a feature optimized to permit the user to create his entire program from top to bottom (of text) without having to move the CP himself.

Moving a tree segment from one location to another is a two-step process. First, the user must "grab" the desired portion of the tree or text. This is done by positioning the CP on the entire portion desired, then entering control-H for "grab" along with a digit from 0 to 9. (Note, then, that the SDE can maintain up to ten "grabbed" segments at a time.) No change occurs on the display, because grabbing a program segment does not delete its present occurrence. The "grab" function is therefore a "copy" function which allows duplication of program segments. To delete the original occurrence, the "erase" command discussed above may be applied after the segment has been grabbed.

The second step in moving a segment is to place it in its new location. This new location must be a nonterminal leaf as described above. Further, the nonterminal must be compatible with the root of the program segment to be attached. Thus, one can not attach a sequence of declarations where a sequence of statements is expected, nor can he even attach it where a single declaration (not a sequence) is expected. The user attaches a grabbed program segment by entering the "put" command (control-X) along with the digit (0 through 9) referencing the grabbed segment. If the segment is not compatible with the CP, an error message will be displayed and the graft will not take place.

The final editing capability of the SDE, inserting, is accomplished through the control-[key, which invokes the "insert before" command. This command is offered only when the CP references an item in a sequence of items in the tree. A sequence is defined in a grammar by the "*", "+", or "..." property of a nonterminal as displayed on the screen. Thus "<decl>*" and "<statement>;..." both indicate

sequences. Entering control-[allows the user to enter an item into a sequence textually in front of the item referenced by the CP. For example, Figure 2.6 shows the display after the user selected control-[when the CP had indicated

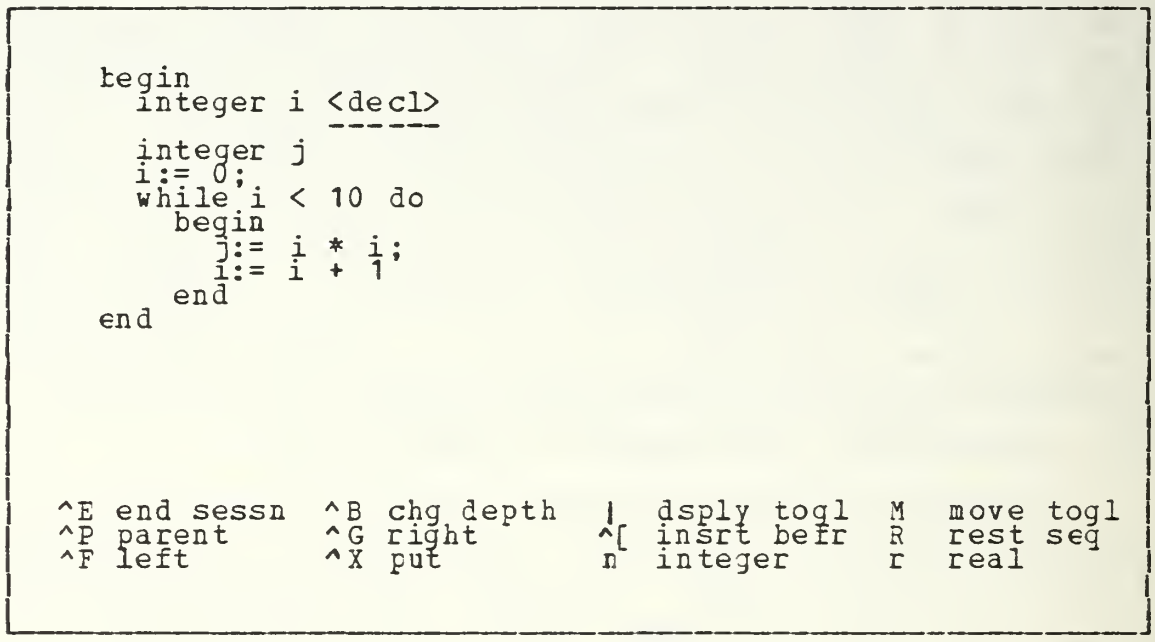

Figure 2.6 Display After Selecting "Insrt Befr"

"integer j". He can now enter a single declaration to be inserted as indicated, using the normal creation commands in the menu.

E. TERMINATING AN EDITING SESSION

The user terminates an editing session by entering control-E. The SDE then asks him two questions:

SAVE PROGRAM IN PARSED FORM (Y OR N)?

SAVE TEXT FORM (Y OR N)?

The first question above corresponds to the "save" or "quit" command found in text editors, for it enables the user either to save the edited tree or discard it. If the user indicates he wishes to save the parsed form, it is saved under the same file name entered during the initialization process; thus the previous version of the program, if any, is lost. If the user instructs the SDE not to save the parsed form, the previous version remains intact.

The second question relates to the text form of the created program. At the user's response to this question, the text form of the program may be saved in a file to be named by the user. Note that this file is textual, and is not suitable for input to the SDE at a later date. However, it is useful in that it can be retained as a text file for archival or inspection purposes. Further, if complete, it represents a syntactically correct program ready for input to a conventional parser or interpreter.

F. ADDITIONAL FEATURES OF THE SDE

While the above capabilities represent a functional syntax-directed editor, the SDE contains several additional features to make it more interactive and responsive to the user's needs. For example, the "display toggle" disables the display of the menu, allowing the user to view more of his program on the screen. A second entering of the command ("|") will restore the menu.

A more significant display feature is the combination "change focus" and "change depth." The "focus node" is defined as that node in the tree at which screen display begins. When viewing the entire program, the focus node is thus the root of the tree. Note that the focus node is always the root of a subtree, and only that subtree will be displayed. The "depth" value indicates how many generations

of descendants from the focus node are to be displayed. Descendants below the depth limit will be displayed by an ellipsis ("..."). The combination of the focus node and the depth limit allows the user to see a detailed view of one portion of his program, or to see an abbreviated view of his entire program, on the display. Figure 2.7 represents a

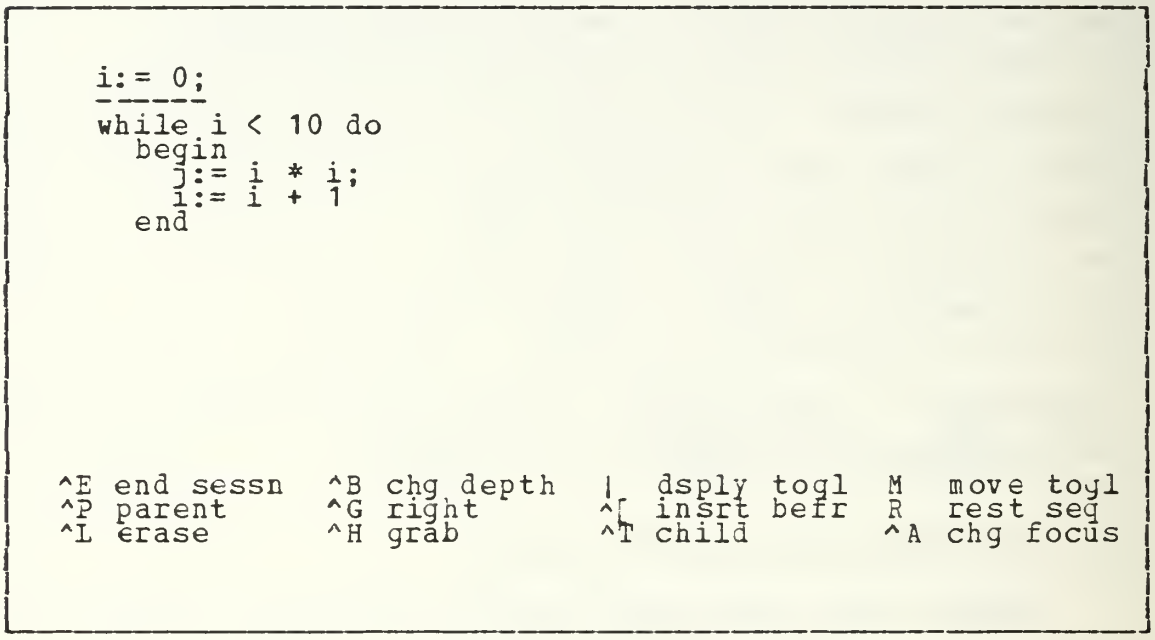

Figure 2.7 Program Display, Focus = Statements

display of the program listed in Figure 2.2 when the focus is adjusted to view only the "statements" portion of the program; the depth limit is set sufficiently high to permit viewing of all aspects of the statements. Figure 2.8, on the other hand, represents a broader perspective of the same program. In this case, the focus node is the root of the tree, and the depth limit has been set low.

A focus node is selected by positioning the CP over the desired program segment to be viewed, then entering control-A for the "chg focus" command. Note that such a

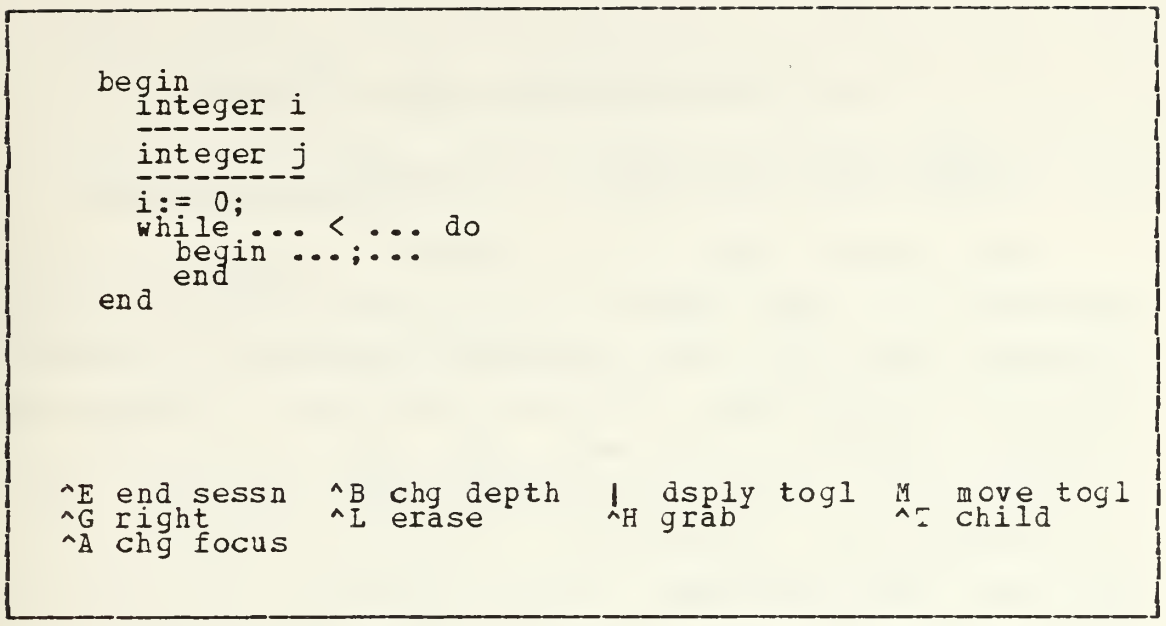

Figure 2.8 Program Display at Low Depth Setting

process can serve only to bring the focus "closer" to the lowest program level. Elevating the focus to a higher perspective is accomplished through the control-P ("parent") command, which automatically raises the focus as necessary whenever the CP is moved upward in the tree. The depth limit is set by selecting control-B for "chg depth," followed by a positive integer.

The final feature to be discussed is the "move toggle." As mentioned previously, the SDE's automatic movement feature is optimized to permit top to bottom entering of text without manually moving the CP. This feature, however, tends to act against the user when editing a pre-established program portion. To inhibit this feature, capital M can be entered. All subsequent movement of the CP must be directed by the user through the movement commands discussed in Section 2.C.

III. THE CONCEPTUAL BASIS OF THE SDE

A. PARSING AND TRANSLATION ON A CONTEXT-FREE GRAMMAR

The textual form of a computer program is written according to the rules of the programming language's grammar, which for most programming languages is classified as being context-free. (Current programming languages often include features that make them more complicated than context-free. Two examples of such features are the requirement for variables to be declared before they are used and the requirement that procedures be declared and invoked with the same number of parameters [Ref. 17: p. 140]. Languages with such features, however, are still considered and treated as context-free, with their special cases handled as exceptions on individual bases [Ref. 18: p. 26].) Formally, a context-free grammar is a four-tuple (N, E, P, S), where N is a nonterminal alphabet, E is a terminal alphabet, E and N are disjoint, S is the "start symbol" and an element of N, and P is a set of productions of the form A --> x such that in each production:

A is an element of N;

x is a string formed by combining any finite number (including zero) of elements from N and E.

The set of terminal strings that can be formed by applying the rules of the grammar is, in effect, the set of programs that can be written using that grammar. A parser is a program that, given a string of terminals, determines if it is a legal program from the language's grammar. The parser does this by finding the derivation (the sequence of applications of the grammar's rules) that would produce the

terminal string. This derivation may be represented (and is usually perceived) as a "derivation tree" whose root is the nonterminal symbol S and whose leaves are the terminals that make up the program. For example, consider the following context-free grammar:

N = {A, T, F};

E = {+, *, (,), q, r, s};

S = A;

P = the set of productions

A --> A + T

A --> T

T --> T * F

T --> F

F --> (A)

F --> q

F --> r

F --> s

The derivation tree of the legal program "q * (r + s)" is shown in Figure 3.1.

While the parser's function is, by definition, the determination of the syntactic correctness of a program, the derivation tree it creates is also an important product in itself. In order for a computer to execute the original program, the program must be translated into a more suitable form known as intermediate code, which can either be interpreted directly or optimized and translated again into machine-executable code (compiled). Program translation is

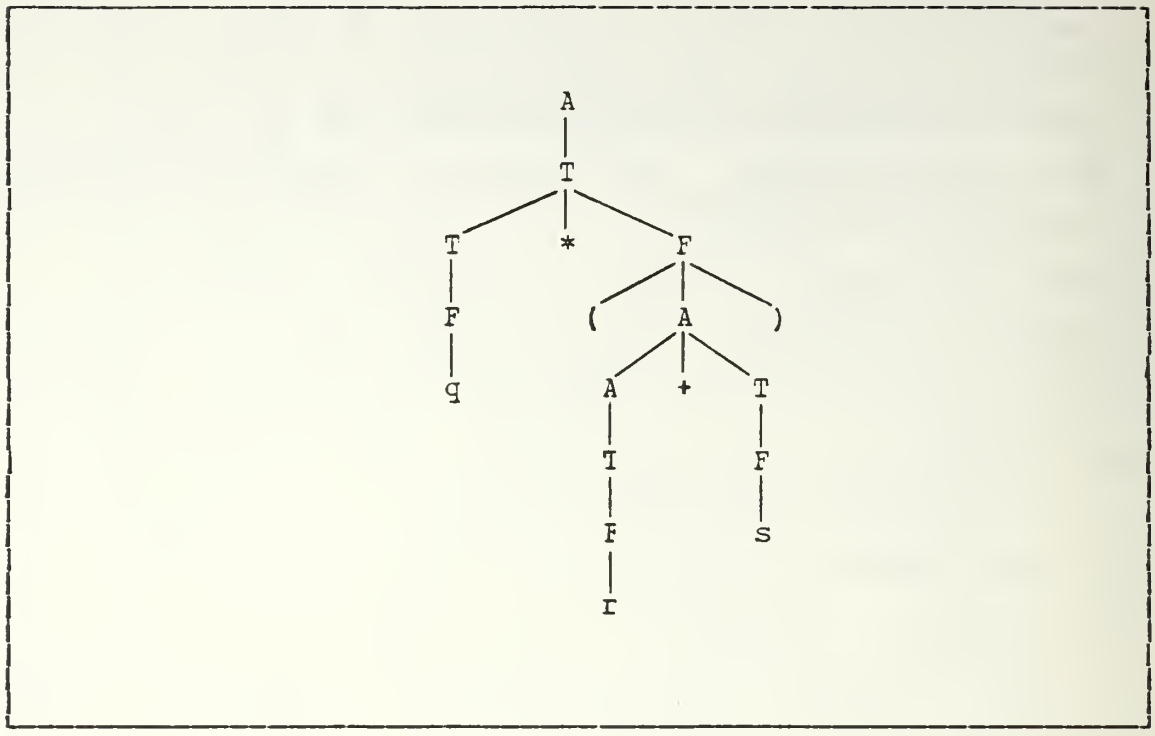

Figure 3.1 Derivation Tree of "q * (r + s)"

accomplished through a "Syntax-Directed Translation Scheme," or SDTS [Ref. 18: p. 279], which conceptually transforms the derivation tree into a "translation tree" by:

1) removing the terminal nodes;

2) permuting the children of each interior node according to a particular translation rule;

3) adding new terminal nodes, members of a new terminal set.

Formally, an SDTS is defined as a five-tuple (N, E, D, R, S), where N, E, and S are the same as above, D is the terminal alphabet of the translation, and R is a set of productions A --> x,y such that in each production:

A is an element of N;

x is a string of terminals from E and nonterminals from N (as above);

y is a string of terminals from D and nonterminals from N;

there is a one-to-one association of nonterminals in x and y.

Note that by following an SDTS, two trees may be constructed. The first is the derivation tree, produced from the "A --> x" portion of the productions. The second tree may be created from the "A --> y" portion of the productions. This second tree is the translation tree, and constructing it in parallel with the derivation tree accomplishes the three conceptual transformations listed above [Ref. 18: p. 296]. In fact, it is the translation tree, not the derivation tree, which is the desired by-product of the parser, for it is a representation of the program's intermediate form.

As an example of program translation, consider the following SDTS, which is an extension of the context-free grammar described earlier:

N = {A, T, F};

E = {+, *, (,), q, r, s};

D = {ADD, MPY, q, r, s};

S = A;

R = the set of productions

A --> A + T, A T ADD

A --> T, T

T --> T * F, T F MPY

T --> F, F

F --> (A), A

F --> q, q

F --> r, r

F --> s, s

The translation tree of the program "q * (r + s)" is shown beside the program's derivation tree in Figure 3.2. Using this SDTS, the translation of the original program is "q r s ADD MPY" (which is the same expression in postfix, or postfix Polish, notation).

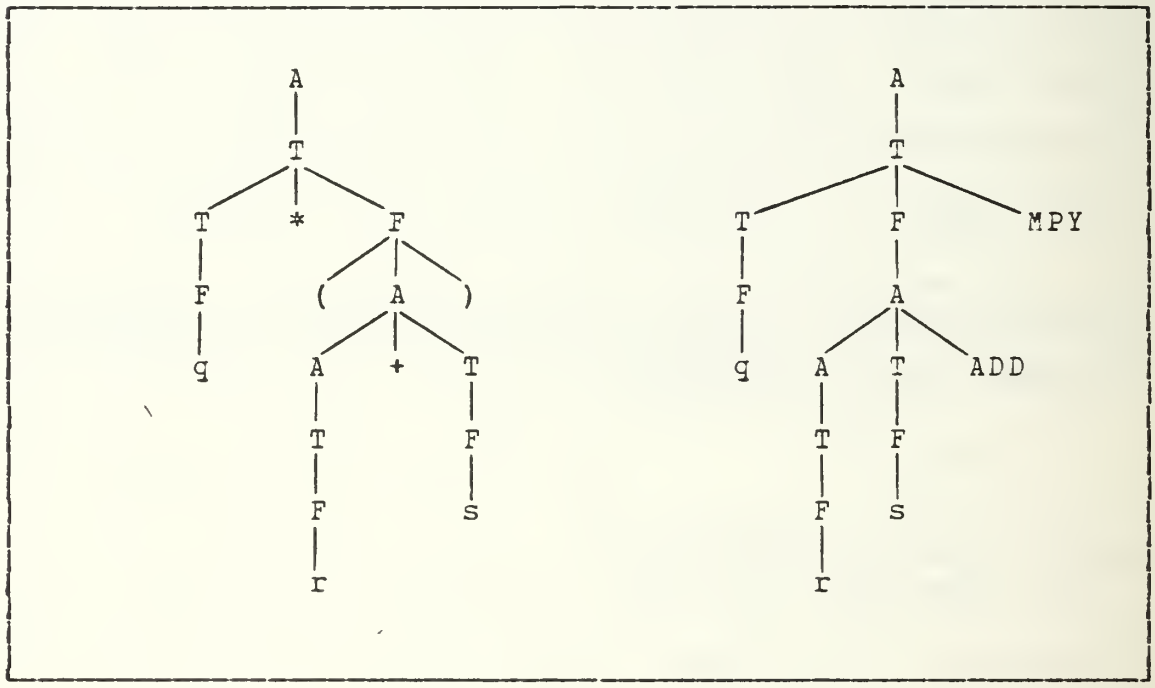

Figure 3.2 Derivation, Translation Trees for "q * (r + s)"

One should note that parsers seldom actually construct the derivation or translation tree as conceptualized above. Some more efficient representation, often involving a stack, is frequently used instead [Ref. 18: p. 46].

B. THE SDE AS PARSER AND TRANSIATOR

As stated above, two major functions of a parser are to determine syntactic correctness and translate the source program into intermediate code. The SDE also performs these same functions, although in a different manner. Syntactic correctness is assured because the editor only creates correct programs. Translation is accomplished by dynamically creating the translatien tree during the editing session.

The SDE creates and edits programs based on an input grammar of the user's choice. The grammar file represents productions in a manner consistent with the above discussion: each production is of the form A --> x,y where x and y are as described above. The SDE, however, uses the elements of x and y in a manner different from that of the SDTS. In the SDE, the "x" portion of a production (hereafter referred to as the "analysis part") determines the textual form of that production in the program as displayed to the user during the interactive session. The analysis part of a rule acts as a template which displays the terminals in a rule and treats nonterminals as "hcles" to be filled in using other rules. The "y" portion cf a rule (hereafter referred to as the "synthesis part") determines what will be added to the tree being created by the SDE when that rule is selected by the user during the editing process.

Intuitively, the SDE only creates syntactically correct programs because the terminals are written by the editor, not by the programmer/user. Whereas a conventional parser uses grammar rules to determine whether a given input string of terminals is correct, the SDE uses the same grammar rules to create the correct input string. For example, based on the sample grammar above, the string "q * r + s)" is illegal because of unbalanced parentheses. A programmer could

erroneously enter such an expression using a text editor, and a parser would detect the error. If the programmer had used the SDE to create the program, however, the parentheses would have been balanced automatically when he selected the rules in the grammar to produce the string he wanted. Note that neither of the parentheses, nor the "*" or "+" operators, would have actually been typed by the programmer at all. The SDE would have displayed these terminals as parts of the templates selected by the user.

When a user creates a program using the SDE, he selects rules to be applied to replace the nonterminal leaves currently in his unfinished tree. As he selects each rule, he instructs the SDE to build onto the translation tree according to that rule's synthesis part. What he sees on his display, however, is determined by the analysis part of the rule. Thus, the programmer dynamically creates the translation tree during the editing session, but need only concern himself with the textual form on the display. Any translation that takes place is hidden from the user.

Actually, the claim that the SDE creates a translation tree is not entirely accurate for several reasons. First, an SDTS allows the translation of terminals from E (the source language alphabet) into other terminals from D (the translation alphabet). While in the example given above the user-defined names q, r, and s translated into themselves, the technical definition of an SDTS does not require this, so they could have been translated into any terminals in D. The SDE, however, stores the actual user-defined names from E in the tree it creates; no translation is performed on them. (The SDE is capable of performing such translation, but its implementation is inefficient and its format proves exceptionally tedious to the grammar writer.)

Another reason it is inaccurate to claim the SDE creates a translation tree is because the SDE is in fact a general

purpose structure editor and not simply a program editor. The term "translation tree" implies that the tree contains information to be used further in a compilation or interpretation process. While the SDE can certainly be used to form such a tree, it is not limited to these applications. The purpose and structure of the tree produced by the SDE depend on the intention of the input grammar designer -- there might be no "translation" involved. (Chapter 5 provides a thorough discussion of the range of applications of the SDE.)

Finally, the translation ability of the SDE is somewhat limited. [Ref. 17] states that the intermediate code produced from a practical SDTS is usually classified into one of four categories: postfix, abstract syntax tree, quadruple, or triple notation. A simple example of postfix (or postfix Polish) notation has already been provided. The SDE can provide such a translation -- Appendix D, for example, lists an SDE input grammar representing the SDTS used earlier in this chapter. Abstract syntax trees are simplified derivation trees in which the interior nodes are operators and the leaves are operands. The SDE generally can not create a tree simplified to this extent because certain nodes having no semantic value need to be retained for the display information in their analysis parts. (An interpreter or code generator using such an intermediate form would have to be tolerant of these useless nodes.) Triple and quadruple notation are discussed in [Ref. 17]. The possibility of translation to these forms was not explored in preparing this paper.

Based on the above considerations, it is more accurate to say that the SDE creates a tree which may possibly achieve a translation of the source program into an intermediate form useful to an interpreter or code generator. It is always true, however, that the SDE insures syntactically

correct program creation. The textual representation of the tree it creates is therefore acceptable as error-free input to a conventional parser, so the SDE is a useful editor regardless of the external value of the tree itself.

It should be noted that the SDE's translation facility, however limited, represents a major difference between the SDE and editors such as that described in [Ref. 16]. These editors create a representation of a program's derivation tree, not its translation tree. It may informally be argued that the SDE is at least as powerful as such editors, for derivation trees result from using grammars whose synthesis parts simply reflect the nonterminals in the corresponding analysis parts.

Finally, note that the use of the term "translation" in this chapter refers to translation from a high-level language to an intermediate form. Translation from one high-level language to another is also possible using the SDE, and this type of translation will be discussed in Chapter 5.

C. A CLOSER LOOK AT INPUT GRAMMARS

As mentioned above, an input grammar to the SDE is a series of rules of the form A —> x,y where x is the analysis part and y is the synthesis part of the rule. The SDE accepts these rules and organizes them into a list of records, each of whose members contains the following fields:

name: the name of the rule (and the nonterminal being replaced);

analysis part: an ordered list of the terminals and nonterminals to be displayed;

synthesis part: the name of the node to create in the tree, with an ordered listing of the sons of that node. A node may have both nonterminal sons and terminal sons. (Note, however, that terminal sons represent terminals of the translation, not terminals in the textual program);

nonterminal dictionary: an index that relates the nonterminals in the analysis part to the nonterminal sons in the synthesis part.

The above components may best be understood through an example. Consider the rule "A --> A + T" from the sample grammar in Section 3.A, but assume its translation is to be "T A ADD" (which represents a reversal of the order of the operands). This rule must be redesigned for input to the SDE as follows. Note the use of quotes to mark terminals and parentheses to mark nonterminals in the analysis and synthesis parts:

```
name: A

analysis part: (A) " + " (T)

synthesis part:

  node to be created: A

  sons of node are:

    1) path: opnd2     expected node: (T)

    2) path: opnd1     expected node: (A)

    3) path: oprtr     expected node: "ADD"
```

The SDE will accept this rule as input, adding it to its list of grammar rules after creating for it a nonterminal dictionary containing the information:

```
1) path: opnd1     expected node: A
```

2) path: opnd2 expected node: T

The name and analysis portions of the above rule are straightforward. The synthesis part, when invoked, causes a nonterminal node named "A" to be created in the tree. This node will be given paths to three sons, and the paths will be given unique names to distinguish them from each other. The relationship between each path name and the expected nonterminal node at the end of the path is recorded in the nonterminal dictionary. Paths to terminals are omitted from the dictionary.

The synthesis part of the above rule deviated from the translation in Section 3.A to demonstrate that the order of nonterminals in the analysis part need not be duplicated in the synthesis part. Note that the nonterminal dictionary entries preserve the order of the nonterminals as they appear in the analysis part. Thus, a sequential access of nonterminal dictionary entries will provide access to the nonterminal sons in analysis part order, which is therefore independent of the order in which they are logically stored in the tree.

(Note that input grammar rules for the editor in [Ref. 16] need only contain analysis parts. This editor generates synthesis parts based on an examination of the analysis parts, which is possible because, as mentioned previously, the editor creates a derivation tree, not a translation tree.)

The SDE permits a grammar rule whose left side nonterminal also appears on the right side of the rule. In other words, recursive productions of the form A --> ABC are permitted. (Grammars including such rules, however, must provide alternative productions to apply to end the recursion. This is analagous to the requirement that a context-free grammar produce only finite-length programs.) Further,

a production rule may produce more than one son of the same node type. Thus, productions of the form A --> ABBC are also permitted. In such a case, it is the responsibility of the nonterminal dictionary to distinguish between similar nodes in a rule and provide the correct path to whichever node is requested.

The notation provided thus far, and the two capabilities listed above, are sufficient to show that the SDE supports the set of all context-free grammars less those that contain "e-productions," or productions of the form "A --> e" where "e" represents the null symbol (or put another way, productions whose right sides contain no terminals and no nonterminals). [Ref. 19] states, however, that a context-free grammar with e-productions is equivalent to one without such productions, except in the case where the null string is a member of the set of strings derivable from the grammar. Thus, the SDE supports any context-free language that does not include the "empty program."

The SDE input grammar convention as described above, therefore, is sufficient to handle all useful context-free languages. For ease of grammar design, however, the SDE also supports several common grammatical conventions:

the Kleene "*", meaning zero or more occurrences of a node;

the Kleene "+", meaning one or more occurrences of a node;

the ellipsis ("..."), meaning one or more occurrences of a node separated by a delimiter;

the option ("?"), meaning zero or one occurrence of a node.

The SDE also allows an abbreviated format for collecting productions with common left sides. This convention results

in a second rule type, the "alternation" [Ref. 15: p. 28], with the following structure:

name: same as above;

series of:

choice id: the command the user will input to identify this selection;

analysis part: same as above;

synthesis part: same as above;

nonterminal dictionary: same as above. Note that, as above, the dictionary is generated by the SDE, not provided by the grammar;

display: what the SDE will display in the menu to describe this choice.

The above is more than a simplifying convention. It allows the grammar designer to compose his own "display" and "choice id" fields. It also has important implications to be discussed in the following section.

It should be noted that the SDE requires a strict format for its input grammars which has been avoided in the above discussion. A thorough description of grammar input and storage is included in Chapter 4 and Appendix B.

D. TREE CREATION, DISPLAY, AND NAVIGATION

As mentioned previously, the user manipulates a tree structure created by the SDE but views the creation in textual form. In this section are presented the two algorithms that correspond to tree manipulation and tree display. Tree display will be discussed first, since it introduces concepts needed to understand the tree

manipulation routine. Also discussed in this section is how the user moves about in the tree.

To understand the display of the tree, it is first necessary to visualize the tree itself. Each node of the tree is a record structure with the following fields:

name: the name of that node type;

syntax: a reference to the grammar rule that produced the node;

parent: a pointer to the parent of that node in the tree;

childlist: an ordered list whose members point to the sons of the node in the tree. Each such pointer is uniquely identified by its "path" attribute.

(The above description of the childlist is the first glimpse of the SDE's actual implementation presented in this paper. Whereas the nodes of a tree are usually pictured as having direct pointers to a possibly variant number of sons, the Pascal implementation of the SDE necessitates an expandable linked list of pointers. This implementation detail will remain exposed throughout the following discussion to avoid confusion in Chapter 4, when the full implementation is presented.)

The above record structure is used for all the nodes of a tree, although these nodes may fall into one of three categories:

1) nonterminals: by definition, they have syntax references and sons in the tree;

2) terminals of the translation (such as "ADD"): these nodes have no sons in the tree, nor do their syntax parts reference anything;

3) terminals from the source language (representing user-defined names): these nodes are similar to terminals of the translation, but they contain user-provided terminal symbols instead of grammar-prescribed names.

The tree, therefore, is a structure whose interior nodes are nonterminals and whose leaves are either terminals of the translation or user-provided terminal names.

The displaying of the textual form of a program based on its parsed tree form is known as "unparsing," and may be represented in a recursive algorithm as follows. Assume the existence of a tree as described above. To unparse such a tree, begin at the root node and follow the following steps:

1) If the node is a terminal of the translation, take no action and return from the recursion;

2) If the node is a user-provided terminal, display the terminal and return from the recursion;

3) If the node is a nonterminal, use its "syntax" reference to access the rule that generated the node. Access that rule's analysis part and nonterminal dictionary. Consider each item of the analysis part in order:

a) If the item is a terminal, display it;

b) If the item is a nonterminal, look it up in the nonterminal dictionary to get a path to the appropriate son in the tree. If there is a node at the end of this path, go to it and unparse it recursively. If there is no node here, display the name of the expected nonterminal on the screen -- this indicates a program not fully created.

The above unparsing algorithm is only a brief summary of what the SDE performs. Details concerning the grammatical

abbreviations ("*", "+", and so on) have been omitted, as have display considerations such as indenting, depth handling, and so on. A more thorough explanation is provided in Chapter 4.

The conversion from tree to display has been discussed. What remains is to discover how the user dynamically creates the tree, a process which shall be referred to as "parsing" since this is what a conventional parser would do given a text input.

There are two general categories of user input to the SDE. One category is the set of language-independent or "standard" commands which the user may invoke either to move about in the tree or to adjust a part of the tree already created. They are standard in the sense that (as a set) they are legal at virtually any phase of program development or position within the tree. Examples of standard commands are Move Right, Delete, Grab, and so on.

The second category of input is the set of language-dependent, "special" editing commands which cause creation of new nodes in the tree. They are special in that their appropriateness is strictly dependent on one's location in the existing tree as related to the input grammar. For example, commands to create an assignment statement are not valid when positioned in a "declarations part" of the tree.

While standard commands have been defined as being legal at any location in the tree, note that certain members of this set will be invalid on certain occasions. For example, it is illegal to Move Right to the next son of a node if it has no more sons there. Similarly, it is illegal to access the parent of the root node or descend to the son of a leaf. Such exceptions, however, have nothing to do with the input grammar being used -- they are purely functions of one's location in the tree.

The key to program creation lies in the algorithm to effect the special commands, and to understand this algorithm it is first necessary to introduce the mechanism by which the SDE identifies its "place" in the tree. The current location is determined through a pair of values called "CN" (current node) and "CP" (current path). The CP always references a path from the CN to one of its nonterminal sons. Viewed from the user's perspective as presented in Chapter 2, the CN always references the parent of the node of concern -- which is at the end of the CP. (This is why "CP" was defined as "Current Position" in Chapter 2. The SDE highlights whatever is pointed to by the Current Path, through inverse video or some other terminal-dependent feature. Since this is the only visual indication to the user of his place in the program, the "Current Path" may rightly be called the "Current Position" as well.)

The CN will always reference a node currently in the tree, while the path indicated by the CP may or may not have a node at its end. Note that special commands are only valid if the CP's path has no node at its end -- in other words, something can be added but not overwritten. Recalling that the synthesis part of a grammar rule names a node to be created and an ordered list of uniquely identifiable paths to sons, the kernel of the SDE's creation algorithm is as follows:

1) Access the grammar rule that created the CN node through its "syntax" attribute.

2) Search the nonterminal dictionary of this rule to find the name of the nonterminal associated with the CP.

3) Look up the grammar rule for this nonterminal.

4) Using the synthesis part of the rule just found, create a new node at the end of the CP's path in the tree. Also

to be created is the node's childlist, which will include the identities of the paths indicated in the synthesis part. Further, if any son is identified in the synthesis part as being a terminal, this son is also created; it will have no syntax part or childlist values.

Note that the above algorithm makes no use of the user's input. In fact, if a grammar having only one possible production from each nonterminal were input, user interaction would be totally unnecessary -- but the grammar could only create one program. Any useful grammar involves selections from alternative productions to be applied at particular locations in the tree. In the sample grammar in Section 3.A, for instance, application of the T --> F production results in a different expression than if the T --> T * F production were selected. Note that whereas a conventional parser selects the appropriate production based on the input text, the SDE must allow the user to direct the selection of a production as the program is being created. This is accomplished through the alternation rules in the grammar. The alternation is the only rule that relies on the user's input at all -- all productions with predetermined synthesis actions can be performed automatically. This fact can be utilized to form a new creation algorithm as follows:

1) same as above;

2) same as above;

3) same as above;

4) If the rule just found is an alternation, match the user's command to the appropriate choice. Proceed with Step 4 as above and exit the algorithm;

5) If the rule is not an alternation, follow Step 4 in the earlier algorithm, change the CN to reference the node just created, change the CP to reference the path to the new CN's first nonterminal son, and repeat this algorithm from Step 1.

Note that Step 5 in the above algorithm requires finding the first nonterminal son of the (new) CN. What if this node has no nonterminal sons? Intuitively, such a node should have been created using an alternation rule, and thus Step 4, not 5, should be executed. The reasoning behind this statement is that unless the user had the option to select the particular set of terminal sons from other choices, the nonterminal that produced these terminal sons is, in reality, simply an abbreviation for that sequence of terminal sons elsewhere in the grammar; thus an equivalent grammar may be designed that removes these deviant cases. Appendix C provides rules for grammar design which result in the development of input grammars which avoid such problems.

One additional feature of the SDE should be discussed here: it is possible to specify a rule with no synthesis part at all. Such a rule is called an "identity rule" [Ref. 15: p. 22], and is of the form A --> B where B is a single nonterminal. The advantage of such a rule is the creation of a more compact tree without loss of semantic information, which in turn implies less memory requirements and quicker unparsing by the SDE as well as quicker interpretation or code generation after editing. Whenever such a production is encountered in the above creation algorithm, the nonterminal on the left side is replaced by that on the right and the rule for this nonterminal is used instead. (This is one reason the term "expected node" was used in describing what a path should lead to. If the rule stemming from an expected nonterminal is an identity, then the path

will lead to a different nonterminal node than originally anticipated.) Note that the right side of the production must contain only a single nonterminal. If it were to include any terminal information, such information would be lost during unparsing, because the node's syntax attribute would reference a different rule.

The final matter to be resolved is how a user navigates through the tree. Chapter 2 defined the five legal moves as Parent, Child, Right, Left, and Rest of Sequence. Note that in all cases, the user may only move to a nonterminal node. (User-defined terminals are accessed indirectly by moving the CP to the parent of the terminal, which is a grammar nonterminal.) "Right" and "left" refer to nonterminal brothers as indicated by a rule's analysis part, so that the user may give commands based on what he sees on the screen. (Recall from Section 3.C that a "right" brother in the analysis part of a rule may actually be constructed as a "left" brother as dictated by the synthesis part.) Further, "child" selects the first nonterminal son of a node (again, "first" being determined by the rule's analysis part). Implementation of these commands is facilitated by the structure of the tree nodes, the creation of the nonterminal dictionary for each grammar rule, and the use of the CN and CP to indicate one's current position in the tree. The commands are implemented as follows:

Parent: Access the CN's "parent" reference. This reference becomes the new CN value, and the CP is set to the first nonterminal child of the new CN;

Child: Access the node at the end of the CP and set the CN to reference this node. Set the CP to the path indicated by the first entry in the nonterminal dictionary for the new CN;

Right/left: Access the syntax reference in the CN node and find the CP's path name in its nonterminal dictionary. Retrieve the preceeding (for left) or succeeding (for right) path name from the dictionary, as appropriate. Traverse the CN's childlist until the corresponding path is found, then set the CP to reference this path.

"Rest Seq" is a movement command available only when the CP references an item in a sequence of syntactic constructs. Such a sequence is implemented in a particular way, as described in Chapter 4, and the "Rest Seq" command utilizes that implementation. Discussion of this command is therefore postponed until the following chapter.

IV. IMPLEMENTATION OF THE SDE

A. GENERAL

The following sections present various aspects of the SDE's implementation. First a general description of the SDE's manner of interaction with the user is provided, followed by a description of some of the SDE's data types and primitive operations that act on or use these data types. Each of the SDE's major processes are then presented in detail. Finally, the SDE's program storage and retrieval functions are discussed.

In studying the following sections, the reader will wonder why certain decisions were made concerning the SDE's implementation. Rather than discuss each such decision, this chapter will present the SDE as it exists, explaining trade-offs and decisions only where beneficial to that presentation. Chapter Five, which represents an assessment of the SDE, will identify strengths and weaknesses of the implementation and will suggest improvements where needed.

As a general comment, however, it should be noted that the SDE is more of a prototype than a finished product. As such, it contains structures and procedures that are less than optimal in terms of efficiency. Some, such as the use of a linked list to house the grammar rules, were used because they were logically straightforward or easy to implement in Pascal. Much of the implementation, however, exists because the SDE was constructed based on the concepts presented in [Ref. 15], and the program inherited some of the conventions of that paper. For example, the input grammar format utilizes parentheses to delimit alternations and analysis and synthesis parts of rules, reflecting the

Lisp format used extensively in that paper. Data types which model "a-lists" (or "asscciation lists," collections of attribute-value pairs) and "tagged a-lists" (a-lists each possessing a single "tag" field at the head of the list) also mimic structures found in the original paper, and the SDE's algorithms are similarly based on those found in the reference. Following the approach outlined in [Ref. 15] facilitated the SDE's development. It is expected, however, that subsequent implementations will deviate more from the details of previous work to prcduce faster, more efficient products.

B. INTERACTION WITH THE USER

Interaction between the user and the SDE can be divided into two distinct phases: display of textual information (the output of the SDE) and acceptance of the user's commands (which serve as input to the SDE). At the highest conceptual level, the SDE executes a cycle of displaying the current status of the program and a menu of appropriate commands based on that status, accepting the user's command, and implementing the command. Interaction is thus a serialized process of display and input. These two functions are discussed individually below.

Display of SDE output requires knowledge of the specific terminal type in use. The SDE must know how to activate and cancel inverse video, how to position the cursor at the beginning of the menu, and how to clear the screen and position the cursor at the top of the display. It must also know the number of lines and columns in the display. Because taking advantage of terminal-specific algorithms or capabilities would limit the SDE's terminal independence, knowledge of the required information is instead provided from a user-supplied external file (the "TERM" file

mentioned in Chapter 2) that can be modified or replaced whenever a different terminal is used. The SDE reads the "TERM" file during initialization and forms four linked lists of integers whose associated ASCII codes will, when sent in succession to the screen, cause the display to perform the four desired functions. Thus, to refresh the display, the SDE writes the "clear screen" ASCII sequence to the terminal, followed by the Pascal "writeln" commands that display the program text; if the menu is to be displayed (as determined by the "dsply togl" value), the "move cursor" sequence is then sent to position the cursor on the menu portion of the screen, and additional "writeln" commands display the menu selections.

One advantage of this approach is that any sequence can be stored in the "TERM" file. Terminals that do not support inverse video, for example, may have printable characters in their "TERM" files so that, when the SDE activates the inverse video function, these characters are displayed instead. Cancelling the inverse video may be done in the same manner. (An example of such a display may be seen in Figure 4.1.) Another use of this feature involves the "move cursor" sequence that positions the cursor for the menu display. On some terminals, this sequence may need to include instructions to clear the remainder of the screen in order to erase the previous menu.

The input of user commands to the SDE is complicated by the use of Pascal as a programming language. It was initially intended that the SDE refresh the screen with every character the user input so that he could see the immediate effect of his entry and view a current, relevant menu. However, Pascal input/output requires that a carriage return be entered in order for prior entries to be read by the program. Since entering a carriage return after each command would prove unnecessarily tedious to the user, the

```
begin
  -> integer i
  -> integer j <-
  i:= 0;
  while i < 10 do
    begin
      j:= i * i;
      i:= i + 1
    end
end
```

Figure 4.1 Sample Display for Terminal W/O Inverse Video

SDE presently accepts a string of commands, up to 80 characters in total length, before the carriage return need be sent. The string's commands are processed individually as if entered separately. The list of legal selections is updated after each selection is acted upon, although the screen (including the menu) is updated only after the final command is completed.

C. DESCRIPTION OF SDE DATA TYPES

The following paragraphs describe the data types used by the SDE to store grammar and program tree information. A formal definition of these data types, as expressed in Pascal, is presented in Appendix A.

At the outset, the SDE's string representation must be explained. Because Berkeley Pascal string capabilities are limited, strings are stored as records having a "wrd" field containing the actual characters in the string and a "len" field representing the length of the string. Note also that the "wrd" field utilizes the Berkeley Pascal "alfa" type, which allows a maximum string length of 10 characters.

Grammar rules are stored in a linked list of records, each of which contains the name of the rule (the nonterminal

on the left side of the production) and a pointer to the right side of the rule. If the rule is a simple production (with a single right side), the pointer references a definition containing analysis, synthesis, and nonterminal dictionary parts. On the other hand, if the rule is an alternation (as discussed in Section 3.D), the record points to a second linked list whose members each contain the single-character choice used to select the particular production, the display string used to represent the choice in the menu, and a pointer to the definition of the production. Figure 4.2 represents the first three entries in the linked list of rules in the "Minigol" grammar in Appendix E. Note that all rules, simple or alternation, eventually

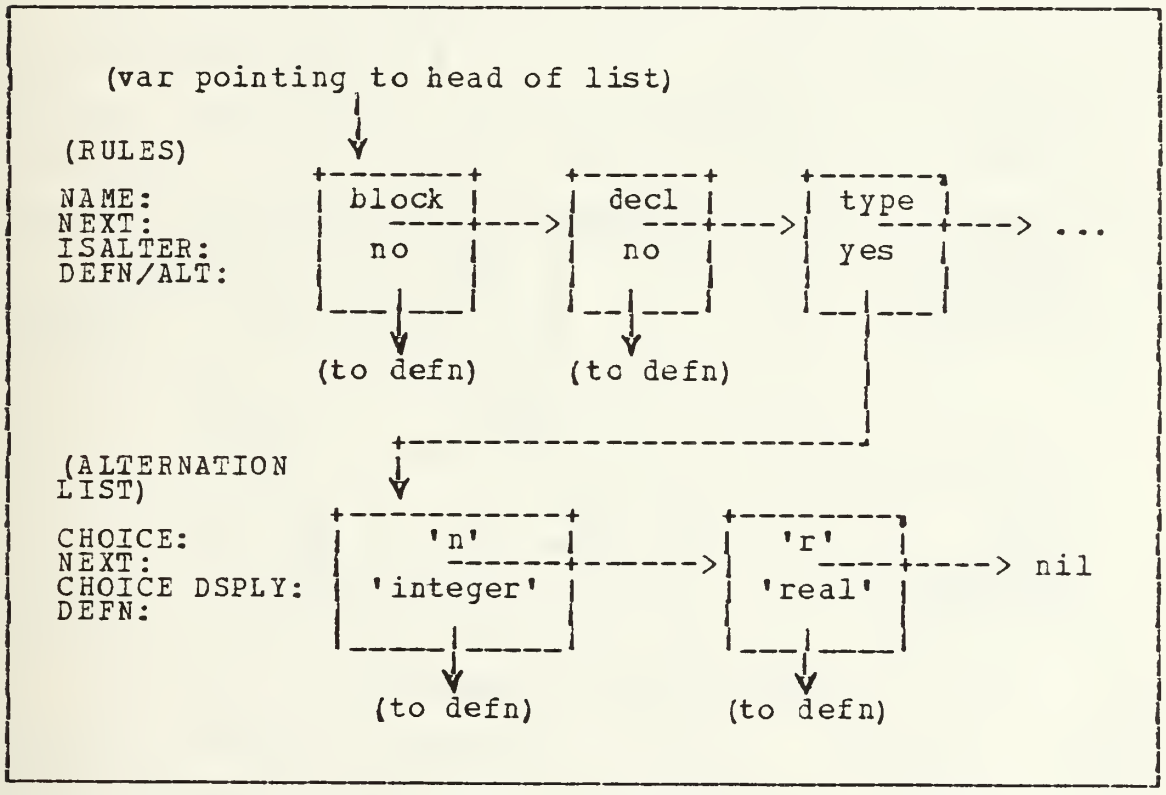

Figure 4.2 Storage of Rules in Minigol Grammar

reference definitions. A representation of such a definition is found in Figure 4.3.

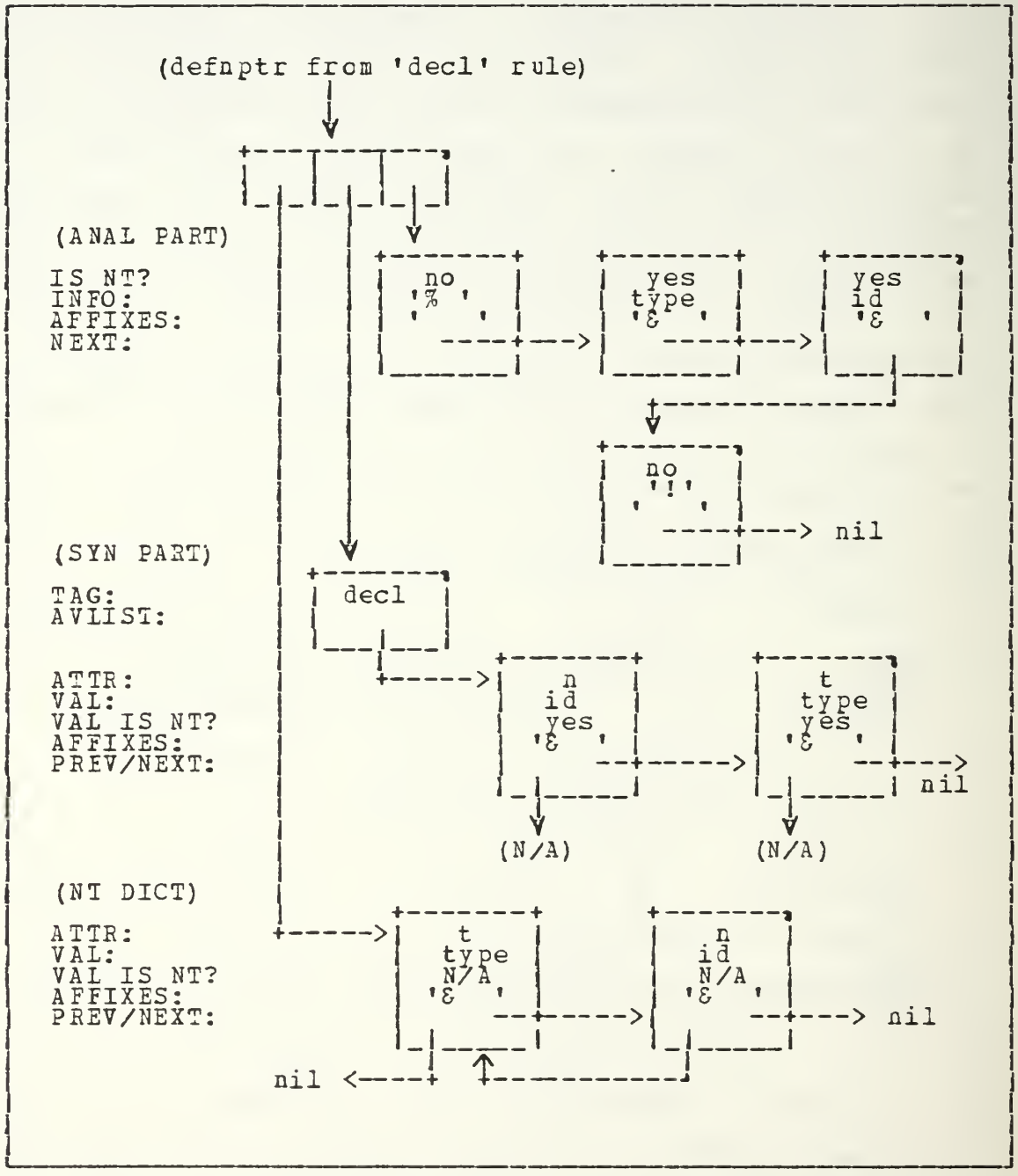

Figure 4.3 Definition Portion of "decl" Rule

A closer look at the definition part of a rule reveals that it consists merely of pointers to analysis, synthesis, and nonterminal dictionary parts. The analysis part is a linked list of records representing terminals and nonterminals. (Note that the "info" field for a nonterminal is the name of the nonterminal, while the same field in a terminal record represents a string of characters, possibly including formatting commands, to be displayed or acted upon during unparsing.) The synthesis part represents a "tagged a-list" (as defined in Section 4.A) whose tag represents the node to be created in the tree and whose attribute-value pairs represent paths to expected children. The nonterminal dictionary is an "a-list" whose entries reorganize the information contained in the analysis and synthesis parts.

Note that all three definition portions contain an "affix" field. This is a record of three characters used to further describe the particular terminal or nonterminal. Nonterminals may be affixed with a Kleene "*", Kleene "+", ellipsis (represented by a single "."), option symbol ("?"), or nothing (represented by a "ε" because the SDE currently requires a non-blank character to be read). If an ellipse is indicated, a second character must be affixed to designate how to separate the individual items in the sequence. Finally, if two or more of the same nonterminal appear in a single production, a third character (a "prime" mark or a digit) must be provided to distinguish between them. (Actually, in such a situation the distinguishing mark is the first of the three affixes.) Terminals need no affixes, but the field is included for simplicity of typing and to prevent accessing nonexistent fields. (While a variant record structure could have been used, it would only have saved the three bytes of storage required by the affixes, at the expense of more complicated logic as well as the space required for the variant structure itself.)

A significant feature of input grammars is the use of the SDE "set" data type to specify the members of an alternation rule whose definitions contain only single-character terminals on the right side of the productions. For example, the production "char --> a | b | c" can be represented as an SDE "set" instead of a linked list of alternatives. Note that definitions eligible for representation as sets would have no nonterminals in their analysis parts and childless tag fields, identifying the selections made by the user, in their synthesis parts. To the SDE, such productions perform no useful function except to record the user's selections. A more efficient means of storing such productions in a grammar (as well as an easier method for the grammar designer to write them) is to list them as the (logical) set of all terminals derivable from the left-side nonterminal. The SDE stores a grammar's sets in a linked list whose records each contain a set name (the name of the left-side nonterminal) and a (Pascal) set of all the derivable terminals.

The above paragraphs describe the data types used by the SDE to store grammar information. The actual program tree is created and maintained using two general record types linked together in a tree structure through use of pointers. The actual tree nodes are "programnode" records having name, syntax, parent, and childlist fields. The children of a programnode are accessed through the childlist field, which points to a linked list of the second record type, the "childnode." Each childnode, in turn, points to a single child (of type "programnode").

Each programnode's syntax field contains a pointer to a definition in the grammar linked list. Note that even if the node were produced from an alternation, the syntax field would reference the specific definition from the list of alternatives. The parent field is a pointer to the node's

parent (in the tree), which is also of type "programnode." The childlist field, as previously mentioned, is a pointer to the first link of the childlist.

The childnode is a record with three fields: path, child, and next. The "next" field provides the linked list structure. The "path" field is a string type and records the name of a path to a child of the parent programnode, as dictated by the synthesis part of the parent's syntax reference. The child field is a pointer to that child, as described in the above paragraph. The children of a programnode are thus grouped through a linked list of childnodes. The particular child in question is accessed by traversing the list until the appropriate path is found.

Figure 4.4 represents a Minigol program tree segment for an integer declaration. The figure includes three program nodes, two of which are children of the third. The two children are accessed through two childnodes. (Note in the figure that the "t" childnode accesses an "int" program node rather than a "type" node, as expected in the grammar rule for declarations. This is because "type" is an example of an identity rule as described in Section 3.D.)

Chapter 3 mentioned the use of "CN" and "CP" to keep one's place within a program tree. Note that CN is a pointer to a program node, while CP is a pointer to a childnode. In Figure 4.4, for example, the "integer" portion of the declaration would be the current position if the CN referenced the "decl" node and the CP referenced the "t" childnode.

It remains to be discussed how the above data types are used to implement the "special" grammar features described in Section 3.C: the sequence conventions (Kleene "*", Kleene "+", and "...") and the optional node ("?"). Sequences are implemented by creating SDE-defined "sequence nodes." These are programnodes named "seq" having exactly

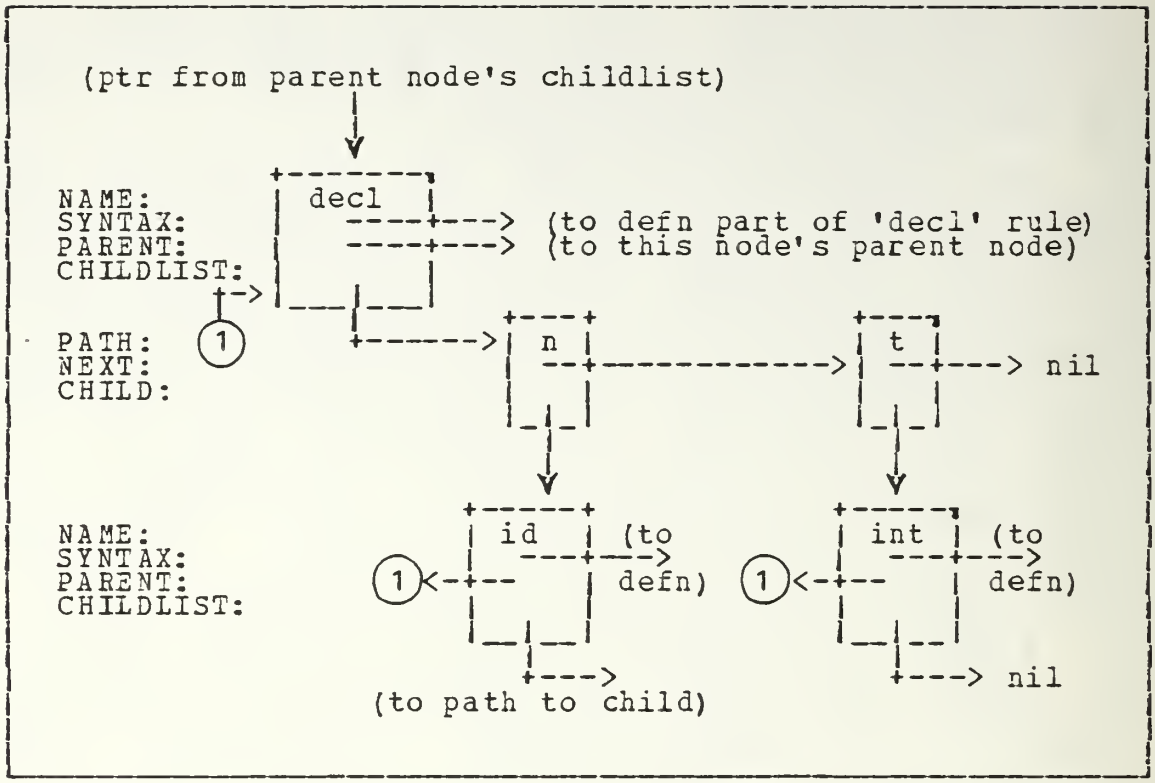

Figure 4.4 Minigol Tree Segment: Integer Declaration

two sons, the first of which is the node type being sequenced, and the second of which is either a "nil" node, which ends the sequence, or another "seq" node. A sequence is thus implemented as a chain of "seq" nodes to each of which is attached one item in the sequence.

The "nil" node mentioned above is a programnode named "nil" having nil references in both its syntax and childlist fields. It is used both to end a sequence and to waive an optional node. For example, suppose an "if" production contains an optional "else" clause. If the user elects to use this clause, then construction continues as if the clause were mandatory. If the user chooses to omit this clause, however, the childnode representing this path is not destroyed; rather, a "nil" node is created and referenced by

the childnode. The parent programnode thus retains paths to all of its children as dictated by its synthesis reference, even if some of those children are not used. Note that the absence of a "nil" node at the end of a childnode is interpreted as an incomplete tree segment by the SDE.

Figure 4.5 demonstrates the use of both "seq" and "nil" nodes. Note that the syntax references of the "seq" nodes refer to the rule that generated the sequence, i.e., the rule that created the parent of the entire sequence. Note also that the first path name in the "seq" node's childlist is inherited from this same rule. The second childnode's name, however, is always "next."

Finally, user selections from a grammar set are stored in a unique way. Sets, as stated above, are useful in a grammar to note the characters that can correctly compose such program entities as variable names, numeric constants, and subroutine names. Because the SDE checks only for syntactic accuracy, any character sequence is valid if syntactically correct -- that is, variables need not have been declared, scoping rules do not apply, and type clashes are irrelevant. In short, the only important information contained in a name is the name itself. It would prove wasteful of storage to allocate a separate node for each letter in each variable name. The SDE therefore takes a different approach by creating a "str" node. Unlike "seq" or "nil" nodes, a "str" node is not given the name "str," but rather assumes the name of the particular set in question. This node has a syntax value of nil and at least one childnode. The first ten characters in the variable name are stored in the "path" field of this childnode, and additional childnodes are created and linked together as needed to house the full name. Note that no programnodes are constructed at the end of any of the childnodes. The value of the childnodes is in the path names themselves.

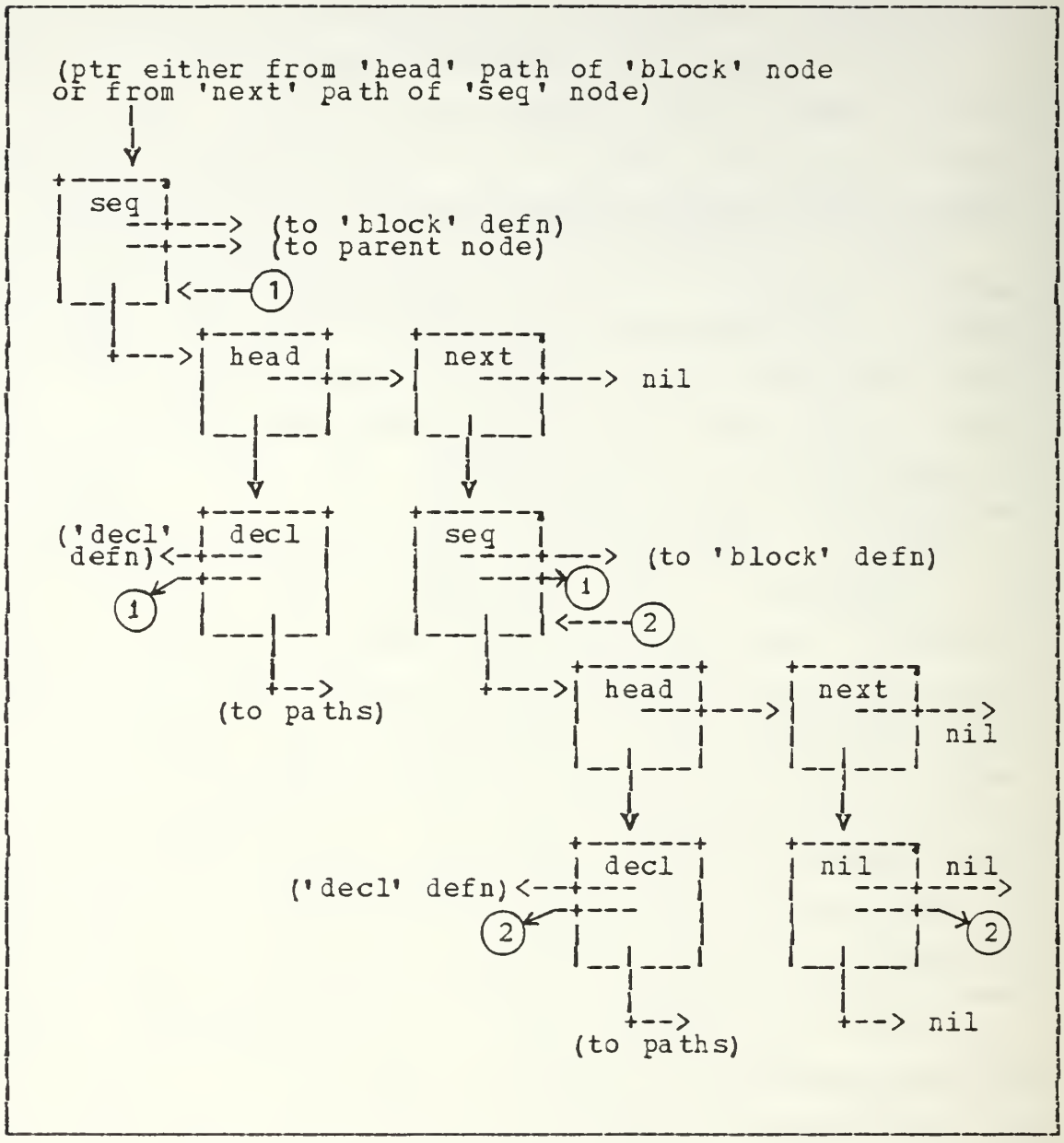

Figure 4.5 Sequence of Declarations

D. SOME IMPLEMENTATION PRIMITIVES

In this section are presented some functions and procedures used by the SDE to peform "primitive" operations. Their understanding both facilitates explanation of higher-

level operations and provides insight into the organization of the SDE as a program.

"Findrule" is a function that, given a name, checks the list of grammar rules to see if it includes a rule by the given name. If it does, a pointer to the entire rule is returned; otherwise a nil value is returned. "Findset" is a similar function that tries to match the argument with an entry in the list of grammar sets. The purpose of these two functions is to retrieve the rule or set entry in a grammar when only a name is known. The use of the nil value also serves as a Boolean indication that the name is not in the specified list.

"Lookup" is a function whose arguments are a pointer to a program node and the name of one of the node's childpaths. The function uses the node's syntax reference to access the definition that generated the node, then matches the name of the childpath with an entry in the nonterminal dictionary. The function returns a pointer to this dictionary entry. "Lookdn," on the other hand, is called with a pointer to a programnode and a dictionary entry as arguments. The function traverses the node's childlist to find a match between a childpath name and the name in the dictionary entry, and returns a pointer to the childnode thus found. These two functions are used in a variety of ways throughout the SDE, both individually and together. For example, when moving to a "right" brother, "lookup" returns the dictionary entry of the present CP; the entry's "next" field is accessed to obtain the right brother's path name; and "lookdn" is used to move the CP to the appropriate childnode.

The last set of primitives to be discussed are those that create nodes in the program tree. "Makenode" is a function that returns a pointer to a new programnode (which is created as a "side effect" by the function). To make the node, the function is provided a pointer to a definition in

the grammar and the value of CN. It creates a new node with a name as determined by the "tag" of the dictionary's synthesis part, a syntax pointer to the definition itself, a parent pointer to the CN, and a list of childnodes as indicated by the attribute-value pairs in the definition's synthesis part. Any terminal sons, as indicated in the synthesis part, are also created at this time.

"Makeseq" is a function that creates a "seq" node with exactly two childnodes. Since the syntax and pathname values are determined by the parent of the sequence and not by the expected child, no definition pointer is required as input; instead, "makeseq" is called by providing the CN and CP values. The CN provides the "seq" node's parent reference, as in "makenode." If the function is creating the first "seq" node in a sequence, the CP provides the pathname which will be copied in the "seq" node's first childnode; otherwise, this information is copied from the parent ("seq") node's first childnode, and the CP is ignored. The result of this process is that the syntax reference and pathname provided by the original parent are passed down through the sequence as new "seq" nodes are created. Identifying what type of sequence one finds himself in is therefore a simple operation.

"Makenil" is a function that returns a pointer to a new node named "nil." The new node is also given nil values in its syntax and childlist fields. The purpose of this function is to create a node that "ties off" sequences or optionals that are not acted upon. Its single argument is the CN value.

"Makestr" is a function that creates a programnode to be linked to a string of characters as specified by one of the sets in the grammar. The function is invoked by providing a pointer to an attribute-value pair (representing a child of the "tag" field node) in a rule's synthesis part; it returns

a pointer to a newly created node. The new node is given the same name as the "value" portion of the input pair -- note that it will always be the name of a set in the grammar. The new node is given a nil syntax reference, like the node created in "makenil," but is also given one childnode, the path name of which is initialized to an "open" symbol followed by blanks. (Later, as characters are added to the string, they will be placed in successive locations in the pathname of this childnode, and the "open" symbol will be pushed toward the end of the word. As the last character in the word is occupied, a new childnode will be attached to the previcus one, and the string will continue in the pathname of this childnode. When the user ends the string with the "endstr" command, the "open" symbol will be changed to a "closed" symbol, indicating the completion of the string.) Note again that unlike "seq" and "nil" nodes, whose name fields contain "seq" and "nil" respectively, a "str" node is given the name of the set whose members it contains in its childnodes. Also, since the user-provided string is stored in the childnodes, the names of these nodes are not dictated by the grammar.

E. DETERMINATION AND DISPLAY OF LEGAL CHOICES

Since the SDE is menu-driven, a mechanism is required to determine what selections should be made available to the user. If this information is retained by the program rather than simply displayed and forgotten, it can also be used to insure the user's input command is legal before it is processed, thereby catching errors early and eliminating the need for error detection and recovery in the command processing portion of the program. For these reasons, the set of legal user commands is assembled into a linked list, displayed in the menu, and retained for comparison against

user input. This process is repeated for each individual command.

The list of legal commands is dependent on one's location within the program tree. Note, however, that the commands to end the session and change the current depth, display toggle, or move toggle settings are always offered. The routine for determining additional commands is as follows:

1) If the CN has a parent value (which will always be true unless the CN is the root node), add "parent" to the list;

2) If the CP has a program node at its end, add "erase" and "grab" to the command list. Further:

   a) If the programnode at the end of the CP has a syntax field pointing to a valid nonterminal dictionary entry (and is therefore not a "str" or "nil" node), add "child" and "change focus" to the list;

   b) If instead the CP points to a "str" node (detected when "findset" returns a non-nil value for the node's name), see if the string is closed; if it isn't, add "any x" (where "x" is the set name) and "endstr" to the list;

3) If, on the other hand, the CP has no node at its end, add "put" to the list and follow the iterative routine in Figure 4.6. This routine searches the sequence of productions that will be applied at this point in the program tree until either an alternation or a set (which is really an abbreviation for an alternation) is found. (Note from Section 3.D that grammar design restrictions insure that one of these two conditions will always be met.) When the alternation or set is found, add the appropriate choices to the command list;

```
{declarations (see Appendix A for types):
   x: alist; (nt-dictionary entries)
   g: grammar; (for rules)
   word: alfarec; (with "wrd" and "len" fields)
   done: boolean;  }

{Note x has already been initialized, using
"lookup," to reference the nt dictionary entry
for the CP path}

word:= x^.val; {EXPECTED NODE NAME}
done:= false;
g:= findrule (x^.val); {RETURNS RULE PTR OR NIL}
while (g <> nil) and not done do
  if g^.isalternation then done:= true
  else begin
    x:= g^.defn^.ntdict;
    if x <> nil then word:= x^.val {WHICH SETS X
                                    TO 1ST DICT ENTRY}
    else {RULE MUST BE AN IDENTITY,
          SO FIND SUNGLE NONTERM}
      word:= g^.defn^.anal^.info;

    {EITHER WAY, WORD NOW HOLDS NAME OF NEXT RULE
    OR SET TO BE CHECKED, SO UPDATE G VALUE }

    g:= findrule(word);
    end; {LOOP BACK TO WHILE}

{Now, either g points to an alternation or is
nil.  If nil, then word holds the name of the
set to use.}
```

Figure 4.6 Routine to Find a Set or Alternation

4) The final commands to be added depend on whether or not the CN accesses a "seq" node. If it does not:

   a) If the dictionary entry has non-nil "next" or "prev" fields, add "right" or "left" to the command list;

   b) If the affixes on the dictionary entry indicate an optional child ("?") or a Kleene "*", and if there is no child presently at the end of the CP, add "endstr" to the command list (to end an empty sequence or waive an option);

5) If, on the other hand, the CN does access a "seq" node:

a) If the CP points to the first son of the "seq" node (i.e., to one of the items in the sequence), then add "right" (which means the next item in the sequence, to the user) and "insert before" to the list. Also add "rest seq" if the second son of the "seq" node exists and is not "nil." Further, if the CN's parent is also a "seq" node, add "left." (The user should be able to travel left and right to items in the sequence without knowing how the sequence is implemented. A "seq" node above the CN means the CP does not reference the first item in the sequence, so there are items to its left on the display);

b) If the CP points to the CN's second son, add "left" to the list. If the CP also has no program node at its end, add "endstr."

Display of the legal choices on the menu is accomplished by accessing each node in the list of choices and displaying the contents of the node. Control characters are unprintable on a screen, so they are displayed as the two-character sequence of "^" and the printable key to be typed on the keyboard. If the display routine detects a set in the list of commands, "any" and the name of the set are displayed.

F. PROCESSING THE USER'S COMMANDS

Every command entered by the user is checked against the current list of legal commands. A command is determined to be legal if it either matches one of the commands in the command list or is a member of the set included in the command list (as discussed in Step 3 of the algorithm presented in the previous section). If the command is legal, it is fowarded to the command processing routines of the SDE. If it is an illegal command, the rest of the

command string is ignored, the screen is refreshed, and a message informs the user that he entered an illegal selection.

As mentioned in Section 3.D, user commands may be categorized as either "standard" or "special." Each command in the command string is first examined by the routine that implements standard commands. If this routine does not act on the command, it is instead processed by the routine that handles special commands. The "stdcmd" routine is therefore implemented as a Boolean function that returns whether or not it acted on the command; any action taken is performed as a "side effect." "Checkspeccmds" is a procedure invoked only when it is already known that the command is not a standard command.

Implementation of the standard commands "right," "left," "parent," and "child" were discussed in Section 3.D. That discussion, however, does not apply when the CN is inside a sequence. In such a situation, "parent" moves the CN and CP to the beginning of the sequence so that the CN references the parent of the entire sequence and the CP references the path that leads to the sequence. The entire sequence is therefore highlighted during subsequent unparsing. "Right" and "left" are implemented so that the CN, CP pair move "down and right" or "up and left" in the sequence structure to reference the next lower or next higher item in the sequence. During unparsing, the right or left item on the display is highlighted. The "child" implementation is unaffected by the presence of a sequence.

The "rest seq" command is implemented by simply shifting the CP to the second son of the CN. Since this command is legal only when the CN is a "seq" node, the CP references its first son, and the second son exists and is not "nil," moving the CP to the CN's second son positions it over another "seq" node. The CP thus accesses a "seq" node whose descendants include the remaining items in the sequence.

"Erase" is another command implemented in two ways depending on whether in a sequence. Normally, the CP's childpointer is simply set to nil, thereby losing the program segment previcusly referenced. (Note that the SDE performs no "garbage collection.") When the CP references an item in a sequence, however, the higher and lower portions of the sequence structure are "tied together" so that only the single link is erased.

"Change focus," "change depth," and "grab" are easily implemented. For "change focus," a variable named "focus" is set to reference the CP's programnode child, and the CN and CP are changed as if the "child" command were entered. Since the "focus" value determines the location in the tree from which unparsing begins, only this subtree is displayed. For "change depth," an integer is retrieved from the command line (or else the user is prompted for one), and this value is assigned to the "depth" variable. For "grab," a one-digit integer is again retrieved from the command line or from the user, and this value is used as an array index to store a pointer to the CP's child.

"Put," the complement of "grab," is a more complicated operation. First, the segment to be implanted must be compatible with the expected node at the end of the CP. Also, the nodes in the accessed segment must be copied and their parent references changed so that the installed segment is completely independent of its model elsewhere in the tree. Compatibility is checked by comparing the root of the segment to be implanted against all the nodes expected at that point in the tree based on the grammar rules. Set compatibility can be checked because the "str" node retains the name of the set whose members it contains. Compatibility of sequences is checked by comparing the node type of the items in the sequence. Duplication of the grabbed program segment is done through a "copy" routine,

which dynamically creates new nodes and gives them all the attributes of the nodes in the grabbed segment except those that form the tree structure (such as parent, next, and so on). The structure fields are given new values that refer to (and result in) the new structure. Note that because no garbage collection is performed when segments are erased, such segments still reside in main memory and are therefore available to be copied (if, of course, they were "grabbed" prior to being erased).

The "insert before" command is implemented in an opposite manner from the "erase" for sequences. Whereas "erase" removes a link from a chained sequence structure, "insert" installs a link into the sequence. Parent and child references of surrounding nodes are adjusted to accomodate the new link.

The implementation of "endstring" depends on whether a user-defined string is to be closed or a sequence or option is to be ended. A string is closed by changing the "open" symbol at the end of the "str" node's string to a "closed" symbol. A sequence or option is closed by putting a "nil" node at the end of the CP.

Finally, the toggle commands ("dsply" and "move") are implemented by negating the previous values of Boolean variables. Using the "display toggle" also adjusts a variable containing the number of lines on the screen. When the menu display is disabled, the unparser plans the display to utilize the larger screen size.

The above paragraphs describe only the standard commands input by the user. The special commands are implemented in an iterative procedure called "checkspeccmds." Note that whereas the routine to determine legal commands (presented in Section 4.E) stepped through grammar rules to find an alternation or set, "checkspeccmds" creates the nodes dictated by the rules along the way to act on the

alternation or set. The coding in this routine follows the algorithm for node creation as presented in Section 3.D, and relies on the restrictions of input grammars as discussed in that section and in Appendix C. Figure 4.7 contains the kernel of the procedure. As mentioned above, "checkspeccmds" continues to add nodes until either an alternation or set selection is found. The user's command is then acted upon, and the procedure terminates.

G. UNPARSING: DISPLAY OF THE PROGRAM TREE

Section 3.D presented a brief summary of the algorithm used by the SDE to convert the program tree to textual form. This conversion is performed by the recursive procedure "unparse," which includes mechanisms to handle sequences and sets. When unparsing for screen display, "unparse" also activates the screen's inverse video (as directed in the "TERM" file) while the CP's descendant programnodes, which reflect the user's "current position," are being unparsed.

As one of its arguments, "unparse" is passed a pointer to a node in the program tree. While this parameter is sufficient to effect conversion of the tree to textual form, two additional parameters are required to insure proper display of the text on the screen. One is an integer representing the current "depth limit," or the number of generations of descendants the user wishes to be displayed. The other parameter is a Boolean value reflecting whether this invocation of the procedure is being made in the first or second "pass" of the unparsing effort, as discussed below.

Unlike a text file, the terminal screen can only display a fixed number of lines at a time. Suppose a terminal displays 24 lines. If the SDE displays more than this number of lines, the first lines will be scrolled up and off the screen. Further, if the menu is to be displayed, the

```
{declarations: see Appendix A for types
   x: alist; (for nt dict entries)
   g: grammar; (rules)
   def: defnptr; (ptr to definition part of rule)
   done, found: boolean;  }

{global variables affected:
   ch: char; (the user's command)
   cn: nodeptr (the current node)
   cp: childptr (the current path)  }

done:= false;
repeat
  x:= lookup(cn, cp^.path); {RETURNS DICT ENTRY}
  g:= findrule (x^.val); {RETURNS RULE PTR OR NIL.
       NOTE G IS RULE FOR EXPECTED NODE AT END OF CP}
  found:= false;
  while (not found) and (g <> nil) do
    if g^.isalternation then found:= true
    else if g^.defn^.syn <> nil then found:= true
    else g:= findrule (g^.defn^.anal^.info);

{AT THIS POINT, G EITHER IS NIL (MEANING A SET WAS
 FOUND), POINTS TO AN ALTERNATION, OR POINTS TO A
 NON-ALTERNATION, NON-IDENTITY RULE WITH AT LEAST
 ONE NONTERMINAL SON.  THIS IS GUARANTEED IF THE
 GRAMMAR WAS DESIGNED PROPERLY.}

  if g <> nil then begin
    {HERE IS FOUND CODE TO CREATE A "SEQ" NODE, IF
     APPROPRIATE.  CODE OMITTED IN THIS FIGURE}
    if g^.isalternation then begin {ALGO. STEP 4}
      def:= findoption (g);
      while def^.syn = nil do begin
        g:= findrule (def^.anal^.info); {IDENTITY}
        def:= g^.defn;
        end;
      {NOW DEF^.SYN POINTS TO THE NODES TO BE ADDED}
      cp^.child:= makenode (def, cn);
      cn:= cp^.child; {THE NODE JUST CREATED}
      if cn^.syntax^.ntdict <> nil then
        cp:= lookdn (cn^.syntax^.ntdict, cn)
        {I.E., SET CP TO FIRST NONTERMINAL SON}
      else moveup; {OR FIND NEXT OPEN SPOT}
      done:= true; {EXIT THE REPEAT LOOP}
      end
    else begin {NON-ALTERNATION RULE, ALGO STEP 5}
      cp^.child:= makenode (g^.defn, cn);
      cn:= cp^.child;
      cp:= lookdn (cn^.syntax^.ntdict, cn);
      end;
    end;
until (g = nil) or done;


{AT THIS POINT, EITHER AN ALTERNATION HAS BEEN FOUND
AND ACTED UPON, OR ELSE A SET WAS FOUND.  IF A SET
WAS FOUND, IT IS HANDLED IN SUBSEQUENT STATEMENTS.}
```

Figure 4.7 Portion of 'Checkspeccmds'

bottom few lines of the display are unavailable for program text. Finally, whatever is displayed on the screen must include the CP's descendants for the menu to be meaningful and for the user to view the program portion he is manipulating. The SDE solves these problems by making two passes through the unparser. In the first pass, the unparser converts the program tree to text but does not send it to the screen. Rather, it records the number of lines between the first line to be displayed and the beginning of the display of the CP's descendants, then calculates how many lines must be unparsed but not displayed so that, when display does commence, the CP's descendants will appear on the center line of the usable screen area. During the second pass of the unparser, text is sent to the screen only after the calculated number of lines is unparsed. Likewise, unparsing terminates after a prescribed number of lines is displayed (to avoid scrolling).

The SDE allows the user to select the depth of his display. As stated earlier, he can use this feature to view the entire program from a broad perspective or inspect a small portion in its greatest detail. The feature is implemented by passing a depth parameter to "unparse," which decrements the value upon receipt. When "unparse" calls itself recursively, it passes the decremented value. A non-positive value causes the procedure to display only "..." and return; no further unparsing is performed on this tree branch. The effect is that unparsing proceeds to the desired depth and no further. Note that sequences are all considered to be of the same depth; thus a degradation of each succeeding item in the sequence is avoided.

Sequences occur in a tree when a grammar rule specifies a nonterminal with an affix of "*", "+", or ".". Unparsing the first two cases is performed simply by ignoring the "seq" nodes and recursively unparsing their two sons. In

the case of the ellipsis (with affix of "."), the first son of the "seq" node is unparsed, the character used to separate the sequence items is printed (if there are more items in the sequence), and the second son (another "seq" node) is unparsed.

Note that the entire tree need not be unparsed every time the screen is refreshed. Rather, only the visible portion of the tree has to be unparsed. Unparsing therefore begins at the focus node, not necessarily the root node, and terminates when depth limits have been exceeded or the screen is already full.

"Unparse" uses three formatting instructions from the input grammar. The grammar designer may specify formatting to effect "prettyprinting" by placing these single-character instructions in the terminal sequences of the analysis part of rules. The symbol "%" denotes carriage return, while "\" and "!" denote two-space indent and outdent, respectively. Note that "\" has two effects: it immediately causes the printing of two blanks, and also moves the unparser's record of left-most column justification two spaces to the right. The "!" symbol, on the other hand, only has the effect of moving the left justification back two spaces to the left. An example of the use of these symbols may be found in Appendix E.

In addition to the formatting as specified by the grammar, the unparser also forces a carriage return and two-space indent (not affecting the left justification) whenever a line is too long for the display (as recorded in the "TERM" file). In this way the unparser avoids splitting of tokens onto two lines and keeps track of the actual number of lines involved in the display.

H. STORAGE AND RETRIEVAL OF THE PROGRAM TREE

The final implementation detail to be discussed is the storage of the program tree at the end of a session, and the reconstruction of the tree from the stored form during subsequent sessions.

Storing the program tree is done simply by writing the name of each node encountered during a preorder traversal of the tree. Terminals of the translation (as designated in a rule's synthesis part), sequence nodes, and "nil" nodes are preceeded by a quote mark. When childnodes with no nodes at their ends (signifying an incomplete program tree) are encountered, the expected node name is printed, surrounded by the symbols "<" and ">". Strings stored in sets are preceeded by an apostrophe and printed as continuous strings, even if they extend through several path names. The "open" and "closed" symbols are also printed at the end of strings. However, if the string was not closed, the name of the set type is also printed, surrounded by the "<" and ">" symbols. Figure 4.8 shows the stored form of the "Minigol" program presented in Figure 2.2. Note that the letter "K" is used to represent the "closed" symbol; the SDE actually uses an unprintable control character for this purpose.

```
block "seq decl id 'iK int "seq decl id 'jK int "nil
"seq asn id 'iK num '0K "seq while reln lt id 'iK
num '10K block "nil "seq asn id 'jK mul1 id 'iK id
'iK "seq asn id 'iK add id 'iK num '1K "nil "nil
```

Figure 4.8 Stored Form of Sample Minigol Program

Reconstruction of the tree from its stored form is somewhat more difficult than storing the tree. The reading of the first character in each name reveals whether it is a string (the character being "'"), an incomplete tree segment ("<"), a terminal, "seq" or "nil" node (the quote), or the "tag" field of a grammar rule's synthesis part. If it is a tag field, function "findsyn" searches the grammar to find the responsible synthesis part. Using this synthesis part, the SDE constructs a new program segment just as it would during an editing session. It also uses the synthesis part to learn how many sons the tag node has, so "readprogram" can call itself recursively the appropriate number of times.

Encountering a "seq" node in the stored form causes the creation of a "seq" node in the program tree. Because a "seq" node always has two sons, "readprogram" calls itself twice recursively. All other names preceeded by a quote in the stored form are given nil syntax and child references and cause return from the recursion.

Strings are also reconstructed based on the stored information. The name of the "str" node to be created is determined by using "lookup" to determine the name of the expected node at this point in the tree. (The expected node will always either be a set or lead to a set through a series of identity rules.) The pathnames of the "str" node's childnodes are input from the string in the stored file. If the string is open, the subsequent nonterminal enclosed by the "<" and ">" symbols is read and ignored; its only value is to provide documentation to the user, should he wish to inspect the tree's storage.

The only restriction this routine places on input grammars is that the tag field of every synthesis part must be unique in order to guarantee that "readsyn" will find the correct synthesis part. Consider the grammar in Appendix D, for example. There are two productions from the nonterminal

"A", so there are two synthesis parts whose tag fields cause the creation of a node representing "A" in the tree. These nodes must be unique, however, so "a1" and "a2" are used. In this particular case, finding the incorrect synthesis part would have destructive effects since the number of sons differs in the two rules. The program would thus either encounter too few or too many names in the stored file. However, consider what would happen if two synthesis parts had identical tag fields and similar children in order and type. "Findsyn" would find the first occurrence of the tag, and because the numbers and orders of children agree, the tree would be constructed without abnormal termination. The syntax reference of the node created would reference the rule found by "findsyn," with the result that the text unparsed from this portion of the tree would reflect the analysis portion of the first rule, not the one intended. The text form of the program thus would be incorrect and misleading.

On the other hand, the SDE's manner of tree storage and reconstruction gives the editor the potential for translation between programming languages. This potential will be further discussed in Chapter 5.

V. APPRAISAL OF THE SDE AND SYNTAX-DIRECTED EDITING

A. MEETING SDE DESIGN REQUIREMENTS: TRADE-OFFS AND SHORTCOMINGS

The SDE was designed to meet several objectives. It was to be highly interactive to facilitate ease of use, yet be table-driven both to support a number of programming languages and to enable use on any terminal. As a program, it was to be portable from one operating environment to another and facilitate certain modifications a user may wish to make to the SDE. Finally, a general objective of the SDE was to be as easy to use as a text editor: the user should not have to "pay" for the advantages of syntax-directed editing by enduring the clumsiness of the tool.

The SDE has many features that achieve interactivity, perhaps the most obvious of these being its menu. It was felt that users taking advantage of the SDE's multi-language capability should not be required to know the details of any input grammar; menu display was therefore appropriate. However, menus often become burdensome in interactive programs, especially as the user gains familiarity with the program and no longer needs to be reminded of his options. For this reason the SDE permits suppression of the menu, an option which has the added advantage of presenting a larger part of the edited program on the display.

The use of two passes through the unparser involved a design trade-off to enhance interactivity. Only one pass is required to convert the program tree to textual form. However, experience with this approach revealed that the unparser often scrolled needed information off the screen, hiding the program portion being edited and making the menu

obsolete. To combat the situation, the user was forced to repeatedly alter the focus and depth settings to position the display as desired. While one solution to this problem would have been to display only a constant number of programnodes on the screen, there is no correlation between the rule referenced by a programnode and the amount of space taken by that rule in the display. In the "Minigol" grammar, for example, the "type" rule only requires the word "real" or "integer" on a single line and lets other rules continue on the same line, while the "block" rule requires at least two lines all to itself. (This example demonstrates a common, known grammar. One can imagine that a user-designed grammar could contain even greater deviation in space requirements.) A two-pass approach was therefore taken as described in Chapter 4. The trade-off involved is that the two-pass unparse is obviously less efficient than a one-pass effort. Indeed, early work showed that interactivity was noticeably affected by the delay between user input and SDE response, mostly because the SDE kept the focus node (from which unparsing begins) at the root of the tree until the user changed it. This meant that a lot of unparsing was taking place to display only 18 lines of text. Efficiency was improved by modifying the SDE to automatically adjust the focus node closer to the CP (by a heuristic number of nodes) when appropriate. This made the two-pass approach acceptably quick, and actually reduced programming effort in other portions of the SDE by keeping the focus node above the CP in the tree. Note that were this violated, as could occur when moving the CP to the "parent" of a sequence without also moving the focus, the display would not contain the CP at all, and the menu and display would be meaningless to the user.

As mentioned in Chapters 3 and 4, the use of the "command string" is another trade-off forced by the use of

Pascal as a programming language. Ideally, the SDE should act on the program tree and refresh the screen (including menu) with every keystroke the user makes; Pascal, however, does not permit such an input method. Even if it did, however, the refresh rate of most terminals would make this approach unacceptably slow. Entry of a string of commands, followed by a single carriage return, offers faster editing at the expense of outdated menus and text displays. It is offered as a temporary solution until either terminal technology improves or a significantly different display algorithm is developed for the SDE.

Inherent in the requirement that the SDE be interactive is that it allow storage of one's work for subsequent editing. Unless useful to an interpreter or compiler (as mentioned in Chapter 1), the parsed program file is useful only to the SDE itself; the user seeks the text form. The tree need never be stored, therefore, if the user always writes complete programs (which will never be modified) in single editing sessions. Such a restriction on program writing is, of course, unthinkable. The tree is therefore stored to enable progressive development of programs by the user. As mentioned in Chapter 4, however, the SDE's routine to recreate a tree from the stored form requires that the input grammar have unique tag fields in its rules. While this is an acceptable restriction on grammars used only by the SDE, it may have consequences on interpreters or compilers seeking to act on the same stored form.

Pascal does not permit storing of pointer values, so the tree could not be stored in any direct manner. Text storage was chosen because it facilitated user inspection as well as tree recreation. Inspection of the stored form is hindered, however, by the choice of "open" and "closed" symbols at the end of strings. These symbols must be unprintable control characters to prevent a user's input characters from being

misinterpreted. However, as control characters they may be displayed in unpredictable, terminal-dependent ways -- in fact, they may be interpreted as commands to the terminal, altering the display itself. This situation may be easily remedied for a particular terminal by selecting different "open" and "closed" symbols (as explained below), but a universal solution is lacking.

A second requirement of the SDE was that it be table-driven. It has been shown that the SDE is capable of supporting virtually any context-free grammar. The restrictions placed on grammar design in Appendix C limit only the representation of such grammars but do not restrict the class of grammars representable in this form. Appendix B lists the format required by the SDE to input such grammars. Note that this format is itself a context-free "grammar" for the "language" of SDE input grammars. The SDE can therefore be used to write grammars for itself. Appendix B includes the syntax of input grammars in a format suitable for input to the SDE. It was used, in fact, to generate the "Minigol" grammar in Appendix E. As with any "program" written using the SDE, the grammars in Appendices B and E are the text (unparsed) forms created in editing sessions.

The input grammar format is admittedly awkward to the human reader. Certainly better grammar representations exist, such as Backus-Naur form and Argot [Ref. 20]. It is possible to use the SDE to create grammars using these more desirable formats as follows:

1) Write (by hand or using the grammar in Appendix B) an input grammar to the SDE that describes the desired format. The synthesis parts of the rules in this grammar must exactly match the synthesis parts in the grammar at Appendix B (except as discussed in a later section).

2) For every programming grammar to be designed:

a) Initialize the SDE using the product of (1) above as the input grammar;

b) Write the programming grammar during the editing session;

c) Save the PARSED FORM of this programming grammar and terminate the session;

d) Re-initialize the SDE using the Appendix B grammar as the input grammar and the parsed form saved in Step 2c above as the input program;

e) Save the TEXT FORM of the tree just created and terminate the session.

The text grammar created in Step 2e above may subsequently be used as an input grammar to the SDE to create programs in this new grammar.

The key to the above technique is that the stored form created in Step 2c is suitable as an input program when the SDE is using either the grammar written in Step 1 or the grammar in Appendix B. Thus the SDE is used to translate programs written in one grammar into acceptable programs under another grammar. The SDE's translation capability is explained further in a subsequent section.

Terminal independence has been achieved for the most part. Use of the "TERM" file provides a degree of information hiding in that the SDE achieves desired display effects without any knowledge of how they are implemented. Maintaining a set of files in the "TERM" format also enables easy transition from one terminal type to another. The simplicity of the format also facilitates user-adaptation of new terminals. Appendix F lists the procedure to create a "TERM" file.

Terminal independence was not fully realized in a different sense, however. The three symbols "\", "%" and "!" were selected as special formatting commands because they are not likely to be useful symbols to a grammar. However, making them special commands not only renders them unavailable for any other purpose, but also means that any terminal lacking these keys can not be used to specify output formats for new grammars. Like "open" and "closed," however, these symbols may be modified by the user to adapt to his particular environment as explained below. Note that the format commands should be printable characters (unlike "open" and "closed") so that the user can inspect his grammar and effect modifications. A way to avoid the unavailability of the selected keys would have been to use two-stroke commands to direct formatting. The sequences could be common keys without making these keys unavailable to the grammar designer for other purposes. "%R" could represent "carriage return," for example, and both keys could be used in other terminal sequences (except as a pair).

The SDE was developed in a Unix environment under Berkeley Pascal. While the SDE is not claimed to be written in purely "standard" Pascal (if indeed such a language exists), the program is designed to minimize taking advantage of its environment's unique facilities. It makes no direct calls to the operating system, although it does invoke the "argc" and "argv" routines to obtain parameters from the Unix command line (see Section 2.A). These calls, however, are purely for the user's convenience and can be removed from the SDE without affecting the rest of the program. In fact, the Unix environment actually hinders the SDE's interactivity. The reader has no doubt been confused over the selection of such meaningless keystrokes as control-T for the "child" command. Such keystrokes were

selected only because more meaningful commands (such as control-C) are intercepted by the Unix operating system before being passed to the SDE; under Unix, control-C causes abortion of program execution. Indeed, (capital) R for "rest of sequence" was selected because all the other control characters either were taken by the SDE or caused an undesirable effect through Unix. The SDE copes with this problem and at the same time offers extreme modifiability to the user by declaring these command keystrokes as variables which are defined during program initialization. When the SDE is moved to another operating system, the installer can redefine all of the SDE's standard commands to whatever keys he wishes. He can also redefine the "open" and "closed" symbols placed at the end of strings as well as the carriage-return, indent and outdent formatting commands. It is recommended that all of these keys (except the formatting commands) be left as control characters, however, because designation of printable characters renders these characters unavailable for grammar or program design. Note also that changing the "open," "closed" or formatting commands between sessions will render previous grammars or parsed forms unusable.

Finally, the SDE was intended to be as useful as a text editor. Toward this objective, the SDE has had mixed success. There are many things a syntax-directed editor can do that a text editor can not. In fact, as mentioned below, a syntax-directed editor even offers certain text-editing facilities not found in a text editor. However, the present SDE also lacks certain features considered crucial to successful program development. First, it is slower than a text editor in that, in most cases, the time required to enter a program under the SDE will be greater than under a text editor. This is due primarily to the command line, which provides feedback only when the display is refreshed.

A much more desirable facility would be to exhibit the effect of each keystroke immediately on the display, preferably without having to redraw the entire screen. Such a facility, however, would be extremely complicated, may require extensive terminal-dependent features, and may even require smarter terminals than presently available. Another reason for the SDE's slowness is the use of control sequences to effect commands; use of "arrow" or function keys would be better, but they are not offered on all terminals. A third reason for the SDE's slowness is inherent in its one-way mapping from tree tc text: one achieves movement about the display only indirectly by moving through the tree. These movements are often unintuitive and cause a jerky motion on the display.

Aside from speed, another advantage of text editors over the SDE is that the user may enter comments under a text editor; no mention of such a facility has been made in this paper. Comments were not included in the SDE's capabilities because it was felt there are no standard conventions for entering them. Some languages, such as LISP, FORTRAN and assembly language, use a delimiter such as ";" or "C" to indicate that the rest of the line is a comment. Others such as Pascal use beginning and end delimiters such as "{" and "}", and everything in between these delimiters is considered a comment. Some versions of PL/I require comments (enclosed by "/*" and "*/") to start and end on the same line. Additionally, of course, individual users will have their own preferences on how to display their comments within the confines of these conventions. All of this implies that comments may not be made a part of the SDE program but must instead be indicated in the particular grammar file being used. However, the only facility the input grammar format has at present is to include an optional "comment" node in every production in the grammar.

This would greatly increase the size of the program tree and also slow down the editing session (because the user would have to waive most of these optional nodes). One feasible solution would be to include a "comments" command in the set of standard commands, but input the details of displaying such comments in the grammar file, possibly in a special, identifiable production with a different notation. This approach would probably require redesign of the programnode structure to reflect the presence or absence of a comment at this location in the tree. Simply extending the childlist would not suffice because the SDE relies on the grammar-provided knowledge of how many children to expect for each node. Further, it is doubtful whether the comments should be kept as part of the tree at all, particularly if the tree is to be shared with other tools that ignore the comments anyway. Sandewall [Ref. 3] points out that comments may create memory management problems (such as fragmentation in virtual memory systems) if the comments comprise a significant percentage of the text. A more economical implementation might be to maintain the comments in a separate file and include index references to this file in the program tree. However, such a system might perform slowly when required to display comments in the context of the program, such as in unparsing or prettyprinting. Further research on implementing comments was not continued in the SDE development; at present the SDE has no comment capability.

The SDE also lacks a "search" facility. Whereas text editors can perform an exact pattern match between the user's search string and the text, there is no such capability when searching a tree of nodes -- the text exists only on the screen. Related facilities such as "global replace" are also lacking. There are several possible solutions to this shortcoming. First, the SDE can actually unparse the entire tree into a file, then read the file and

perform pattern matching. However, such a strategy would only be able to acknowledge whether a pattern was present, then display this pattern and its surrounding strings on the screen; the SDE would be unable to relate this pattern to a particular node in the tree because of the one-way nature of mapping from tree to text, although perhaps a complicated approach could effect this. The SDE would certainly be unable to effect a replacement of a random string with another because only user-defined names and grammar templates may be syntactically changed. A better strategy would be to provide a limited search capability restricted only to user-defined names, and perhaps to maintain a table of such names with pointers to all occurrences of them in the tree. This would provide rapid response to the search query and permit transactions (such as repositioning the CP or changing the name) to be made on the tree. Searches for random strings such as "for i :=" could not be implemented this way, but searching for the variable "i" would eventually find the desired locations. A third strategy would be to enable the SDE to perform "parsing" of the search string, then to perform pattern-matching between the tree segment thus created and the program tree. Note, however, that none of these solutions offers as great a facility as is presently available in text editors. On the other hand, the second and third approaches offer a certain advantage in exchange for their inflexibility in that they will retrieve only those matching patterns that are also the proper structures in the grammar. Whereas a text editor will retrieve every occurrence of "i" in a program (including keywords such as "is" and "in"), the above methods will retrieve only the variables named "i."

B. IMPROVEMENTS AND EXTENSIONS TO THE SDE

Like most useful programs, the SDE is not a static product. It has evolved since the beginning of this project, undergoing several significant improvements. One such improvement was the use of pathnames to house string information. This modification not only improved storage efficiency and eased grammar design, but it also was implemented within the data structures already established, facilitating the modification as well as program comprehension.

The SDE is therefore a growing, changing product, and certainly it is not perfect in its present state. The shortcomings cited above must be overcome; there are additional modifications that should be made to improve the efficiency with which the SDE presently functions; and further modification should be performed to give the SDE additional features and capabilities.

One efficiency improvement that can be made is the SDE's implementation of sequences. Sequences are presently implemented through chains of "seq" nodes each referencing exactly one item in the sequence. While there are certain advantages in knowing that each "seq" node has exactly two sons, it would be more efficient to link all items in a sequence as sons of the same "seq" node. Moving to right or left brothers or to the parent would be facilitated; however, other commands such as "rest seq" would have to implemented differently. Program tree restoration from the stored form would also have to be altered, since the number of recursive calls to "readprogram" would not be known at the time the "seq" node was read.

Another improvement that can be made in the SDE is to reduce the storage overhead of grammar rules. For example, rules presently use Pascal strings to describe nonterminals

in their analysis, synthesis, and nonterminal dictionary parts. These nonterminals always reappear as names of other rules. A more efficient implementation would be to store pointers to these rules in the dictionary parts that reference them rather than using the names themselves. The overhead of establishing pointers would be paid only once per editing session during initialization. While pointers would add indirection to operations such as displaying nonterminal names from an analysis part, it would eliminate the overhead of string-matching such as in "findrule." Another such improvement would be the use of integers instead of ten-character strings to represent path names in the synthesis and nonterminal dictionary parts of the rules. In fact, such pathnames could be generated sequentially by the SDE for each rule; the grammar designer would not have to plan for them or write them at all. Use of integers would not only be more efficient for the grammar designer and for storage management, it would also improve the efficiency of looking down childlists to find a match between a given path name and a particular childnode. "Lookdn" could simply traverse the childlist the appropriate number of childnodes, as determined by a path's integer "name."

Another improvement to be made in the SDE is the storage of user-provided strings. While the present implementation represents a savings over a character-by-character implementation, it remains wasteful. By making the "childnode" structure a variant record, it could either reference a programnode or a new structure designed especially for string information. The "str" node and the establishment of entire childnode records could be avoided. Another improvement to string storage would be to include a "length" field in the record itself, which would eliminate the present delay in searching the entire string to find the "open" symbol when adding new characters, as well as provide the

unparser's formatting mechanism with the length of the string more quickly.

The need for all tag fields to be unique can be eliminated through application of artificial intelligence techniques. When restoring a tree, the SDE could determine which synthesis part were referenced based on the nature of the attribute-value children of the tags in the rules.

Another obvious modification to improve the SDE's efficiency is to eliminate the use of two record types in the program tree. The childnode only contains a pathname, a pointer to a programnode, and a link to another childnode. The programnode pointer field can be eliminated without loss of information by adding the other two fields to the programnode structure itself. This would save memory space and make the tree structure more understandable. Current SDE use of the CN and CP values, of course, would need revision.

Finally, the SDE's lack of "garbage collection" should be corrected to improve efficiency of memory use. While not disposing of program segments that have been "erased" by the user avoids more complicated logic within the SDE (for example, in the "grab" function, which capitalizes on the present implementation), it can be extremely wasteful if a large amount of editing is being performed. Past experience with the SDE has been limited to operating on a VAX 11/780 minicomputer in the editing of small programs, and system performance has not sufferred under the present implementation. However, the SDE should be made more economical to insure acceptable performance both in editing larger programs and in operating on smaller computers.

In addition to improvements which increase the efficiency of the SDE, there are also some extensions to the SDE that can be made to make it a more useful product in an interactive programming environment. For example, the SDE

can be made a more powerful tool by planning to handle certain semantic information as well as purely syntactic detail. For instance, most programming languages are categorized into only a few groups based on the type of scoping they utilize. SNOBOL, APL, and traditional LISP use dynamic scoping, while PL/I, FORTRAN and the ALGOL family of languages use static scoping [Ref. 17: p. 38]. If the SDE were so informed, it could perform type or scope checking during the editing session, and discover undeclared variables for statically scoped languages. Another example of semantic help provided by a syntax-directed editor would be the checking of parameter lists. Most languages pass parameters either by name, reference, value, or some combination of these methods. Knowing, for example, that Pascal implements both "pass by value" and "pass by reference," a syntax-directed editor could detect the passing of a literal constant by reference (which is a security violation because it risks altering the value of the constant) and so inform the user [Ref. 21: p. 221]. It should be noted, however, that the more such checking is "built in" to the editor, the less likely it is to retain its generality as a "universal" editor.

One class of languages for which the SDE could provide some very useful semantic information is the set of grammars it may generate. The input grammar in Appendix B guarantees the creation of syntactically correct grammars, but this alone does not insure that the grammar produced is usable. It does not insure, for example, that the nonterminals in an analysis part agree in name and affix with those in the synthesis part of a rule, or that all such nonterminals appear as rules or sets elsewhere in the grammar. The SDE presently detects certain grammar errors during initialization, but lacks complete semantic checks and error recovery. Further, it would be more desirable to detect such errors

during grammar creation rather than at attempted usage. To be a better tool, therefore, the SDE should include a routine that insures grammars are correct as they are being created. The semantics of such grammars are very simple, and it is anticipated that such a routine would be small, efficient, and easy to implement.

C. IMPLICATIONS OF SYNTAX-DIRECTED EDITING

This paper has presented an implementation of a table-driven syntax-directed editor. While such an editor is intended for use in creating syntactically correct programs, syntax-directed editing offers capabilities that extend far beyond this function.

The grammar-generating grammar in Appendix B offers a rapid means with which language designers can examine their research grammars. When contemplating a new language, the designer can input his ideas into the SDE, then use it to quickly see a textual example of his grammar. A grammar checker as proposed in the previous section would further insure that his grammar was complete. Further, the SDE can display programs written in an incomplete grammar (up to a point), so that the grammar may be developed incrementally. Finally, the designer may use the SDE to create program trees using his grammar, allowing him to insure that the desired semantic information can be derived from his source language. Lacking a syntax-directed editor, these results could only be obtained after developing a scanner and parser for his language (although parser-generators exist to help in this effort) and then writing complete programs in his new language using a text editor.

While use of a syntax-directed editor as described above facilitates the development of "traditional" grammars, i.e., those restricted to the class of LR(1) languages [Ref. 17:

p. 261], syntax-directed editing implies a far greater range of possible programming languages in the future. In the past, language design has been limited to those languages that can be parsed effectively. Syntax-directed editing removes such a limitation on grammar design, since the editor is told by the user which production to apply at a given point in the program. The program tree is generated directly during program creation, not through parsing a text. The implication is that future languages may be designed to be more readable to the human reader rather than to a parser. Consider, for example, the use of the "where" clause in the applicative expression notation "L where X = M," in which the "where" clause binds variables in L, as listed in X, to values derived from the expression M [Ref. 22: p. 314]. This is difficult to parse conventionally, since binding details are provided after the variables are used and because determination of which production to use requires "look-ahead" of as many tokens as are in L. Such a notation presents no difficulty to a syntax-directed editor, however, since the user provides determinism by directing the application of the intended production. Note also that syntax-directed editing eliminates the need for current lexical conventions, since no scanning need be performed. Thus, a variable named "this is a single variable" is perfectly acceptable when input through a syntax-directed editor.

Even the one-dimensional concept of a program as a stream of characters is made obsolete by syntax-directed editing: two-dimensional languages may be designed, using symbols difficult to parse traditionally. For example, the equations in Figure 5.1 are unacceptable to a traditional parser, yet can be parsed and displayed by a syntax-directed editor with the proper formatting commands. In short, syntax-directed editing opens an entire realm of programming languages previously considered impossible.

$$\text{expected value} = \sum_{i=1}^{k} a_{(i)} \, p_{(i)}$$

$$\text{num of real solutions} = \begin{cases} 2, & b^2 - 4ac > 0 \\ 1, & b^2 - 4ac = 0 \\ 0, & b^2 - 4ac < 0 \end{cases}$$

Figure 5.1 Examples of Two-Dimensional Equations

Syntax-directed editors alsc have an implication in the education of computer programmers. It would be unnecessary, for example, to instruct the syntax of any language. Whereas the Pascal student must currently know that statements always end with a semi-colon, this need not be known by a student using a syntax-directed editor, since the editor installs the semi-colon automatically. The student can thus learn the semantics of the language more readily, being freed from syntactic concerns.

The use of a syntax-directed editor as a translator has already been demonstrated in Section 5.A. There, the SDE was used to translate grammars from Argot into the input format of the SDE. Translation between grammars is possible when both grammars have corresponding rules with identical tag names and child patterns in their synthesis parts. They need not be completely identical, however. Note, for example, that there is a slight difference between the grammars in Appendices B and H in that the second alternative production of the "rule" alternation has a "+" affix on the "choice" son in Appendix B, while the affix is ".;" in

Appendix H. Since both are represented identically in a tree, this difference is unimportant. Note that even the word "choice" need not have been duplicated, so long as the new name referenced a rule that had the same tag name and child pattern.

Translation is also possible between higher-level languages, although such translation depends on the nature of the two languages themselves. It should be possible, for example, to translate a subset language program into a superset language program. Some translation within a "family" of languages should also be possible. Appendix I lists modifications to the Minigol grammar to make it compatible with the Pascal subset grammar of Appendix J, which may be perceived as a grammar for Pascal functions and procedures in that it allows declarations of variables before a master "begin-end" block. Using these two grammars, the Pascal program segment in Figure 5.2 may be trans-

```
var
  i: integer;
  j: integer;
begin
  i:= 0;
  while i < 10 do
    begin
      j:= i * i;
      i:= i + 1
    end
end;
```

Figure 5.2 Pascal Version of Figure 2.2

lated into the Minigol program segment in Figure 2.2 simply by storing the parsed form under Pascal and reconstructing the tree under Minigol. The nature of this translation bears closer examination.

The Pascal grammar in Appendix J has similar synthesis parts to those of the Minigol grammar in Appendix I with one notable exception: the "block2" rule. In the Minigol grammar, the synthesis part of this rule creates a "block2" node with paths to two nonterminal sons, the "decl" sequence and the "stateblock" node. In the Pascal grammar, however, this synthesis part creates paths to a nonterminal "stateblock" node and a terminal named "nil." This is because Pascal blocks do not include a "declarations" part while Minigol blocks do. (For example, Minigol would allow declaration of additional variables above the "j:= i * i" statement.) The interesting relationship between these grammars is that any program written in one may be displayed (or translated) using the other grammar. In the curious case of translating a Minigol program (with declarations in its blocks) to Pascal, the declarations simply disappear. This occurs because the "readprogram" routine in the SDE recursively constructs the tree from the stored form based only on the expected number of children for each tag according to the synthesis part that generated the tag. Since the child patterns are the same in both grammars, "readprogram" reads the correct number of children and places them correctly in the tree when it encounters them. In this particular case, however, instead of reading a terminal named "nil," the routine encounters a sequence of declarations, and since declarations have their own rules in the grammar, the routine continues to process the file. In other words, the declarations are still present in the Pascal-constructed tree. The reason they are never seen by the user is because unparsing is dictated by the analysis, not synthesis, parts of the rules. Thus, since the analysis part of the "block2" rule in Pascal does not mention a "decl" sequence, it is not listed in the nonterminal dictionary for that rule, and the unparser bypasses this child branch as if it were a terminal

of the translation. The implications of this invisibility are that the Pascal grammar should be treated as a subset of the Minigol grammar, even though the two are not usually considered this way, and that programs should never be translated into Pascal from Minigol. Such a translation would risk causing other tools working directly on the tree to encounter inappropriate nodes; even tools acting on the text form would encounter difficulty, since variables declared in the (Minigol) blocks would have never been declared in the (Pascal) text form of the program.

Still another type of translation offered by syntax-directed editing is translation between representations of the same language. [Ref. 23], for example, describes four "alternative syntactic forms" for a particular object-oriented language. Because all four share an indentical program tree format, translation from one form to another is a simple matter of reading the stored form using the grammar of the desired representation. Note that translation between forms of the same semantic language does not involve the problems mentioned above. However, since one of the four forms is two-dimensional, it may require special formatting commands to display a text program under this grammar.

All of the above capabilities of syntax-directed editors are not reachable by standard text editors. Text editors can not translate, check syntax, or do anything to insure the correctness of a program. What is interesting, however, is that a syntax-directed editor may even offer text-editing capabilities not found in text editors. For example, the structure imposed on text by a syntax-directed editor should enable the user to view any text top-down, stopping at a particular depth, so that, for example, only section titles (but not section contents) are displayed. In fact, this type of structure serves to provide an automatic table of

contents for the document. Another application of user-selected viewing is in the use of comments in a program. It should be possible to structure comments into levels of complexity so that a reader unfamiliar with the source code may view only the high-level comments to get a broad view, while readers thoroughly familiar with the project can view a more specific, detailed set of comments. A variation of user-selected viewing is creator-designated security. There may be certain items (text, code, etc.) that are not meant to be viewed by certain classes of user. These items need not be removed from the product, but need only be "filtered" from view by not displaying them.

Syntax-directed editing is appropriate for creating and manipulating anything representable hierarchically. One obvious candidate for such editing is a hierarchical database. It is relatively simple to prescribe a "grammar" for a hierarchical database, which may be pictured as a collection of trees whose roots may have any number of dependents, each of which may have any number of dependents, and so on [Ref. 24: p. 67]. Appendix G, for example, lists an SDE input grammar that produces a tree of information about an organization's training course history given the following hierarchical structure:

> each course has a number, title, description, list of prerequisites, and list of current offerings;
>
> each prerequisite includes a course number and title;
>
> each offering includes a date offered, location, and format, as well as lists of instructors and students;
>
> each teacher has a number and name;
>
> each student has a number, name and grade [Ref. 24: p. 280].

```
Course: M23  Title: DYNAMICS  Descr: ....
  Prereqs:
    Course #: M19  Title: CAICULUS
    Course #: M16  Title: TRIGONOMETRY
  Offerings:
    Date: 750106  Location: OSLO  Format: F2
    Date: 741104  Location: DUBLIN  Format: F3
    Date: 730813  Location: MADRID  Format: F3
      Teachers:
        Emp #: 421633  Name: SHARP, R
      Students:
        Emp #: 761620  Name: TALLIS, T  Grade: B
        Emp #: 183009  Name: GIBBONS, O  Grade: A
        Emp #: 102141  Name: BYRD, W  Grade: B
```

Figure 5.3 SDE Representaticn of Hierarchical Database

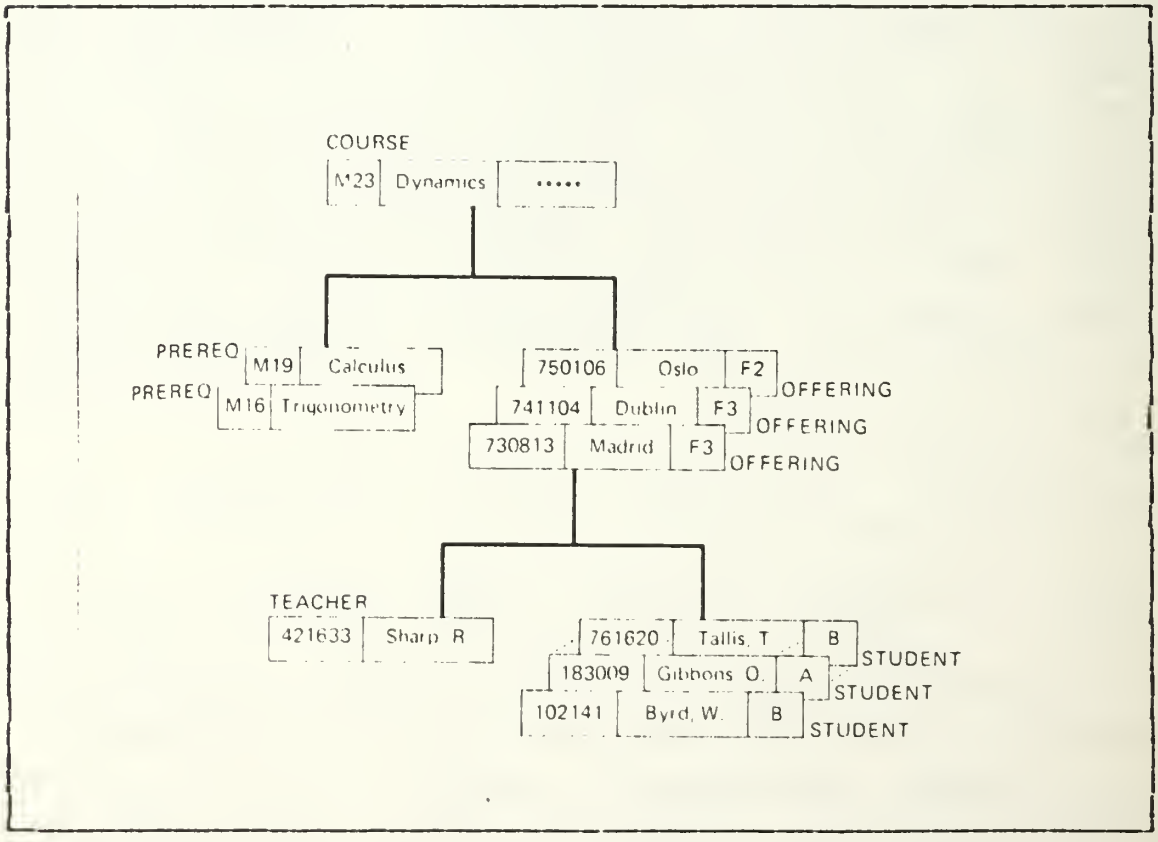

Figure 5.4 Hierarchical View of Database

A sample display of a database created using the "grammar" at Appendix G is shown in Figure 5.3. Figure 5.4 contains a representation of the database as pictured logically in [Ref. 24: p. 281]. Not only does one note that both figures represent the same information, but one can also imagine the similarity in structure between Figure 5.4 and the tree constructed by the SDE to represent the database. This discussion does not intend to suggest that a syntax-directed editor is capable of performing database operations. It does show, however, that such editors can represent hierarchically structured data. It is further suggested that such editors might serve as suitable editing devices to create and modify databases representable in a particular hierarchical structure.

D. CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

The overall objective of this paper has been achieved in the SDE: a working table-driven syntax-directed editor has been implemented, and many lessons regarding syntax-directed editing have been learned through this implementation. It has been shown that the SDE can support virtually any context-free grammar. As a program, it has been implemented to be as portable and flexible as possible through the use of variable command keys, the "TERM" file, and avoidance of system-specific calls. In its 1600 lines of Pascal code, it performs the functions of both parser and prettyprinter and creates, stores, and retrieves files. These accomplishments having been noted, the SDE must now be appraised on its effectiveness as a useful product.

The SDE is too slow for actual program development. The compromise of the command line and the need to refresh the entire screen, rather than only the edited portion, cause an unacceptable delay between user input and program response.

Simple navigation about the tree is clumsy and unintuitive. In short, the user interface needs significant improvement.

The SDE also lacks features essential to a viable editor. Lack of any search mechanism is a severe limitation. The absence of a comment input mechanism has also been cited.

As for display, the current SDE still has problems forcing line breaks on long display lines. When editing grammars, for example, it occasionally splits terminal strings after the opening quote, with the result that prettyprinted forms are perceived to have long strings extending over two lines. Another problem is that the heuristic movement of the focus above the current node, done to increase efficiency and reduce the need to change depth and focus manually, causes "tunnel vision" when editing the last items in a long series. For example, attempting to edit the last terminals and nonterminals in the rules in Appendix G causes only the concerned rule (in fact, sometimes only a portion of the rule) to be displayed.

As a prototype, the SDE has done its job in identifying these shortcomings and providing lessons from them as well as from its successes. Further, the shortcomings cited above provide ample subject material for future research. It is felt that the most significant potential for improvement to the SDE lies in its interactivity with the user. Direct editing on the CP rather than through a command line, accompanied by immediate update of text and menu alike, would help make the SDE a viable, useful editor.

Aside from continuing the development of the program itself, the potential of the SDE as it already exists has yet to be explored. It has been suggested that the SDE has certain translation capabilities as well as the ability to create an executable program tree. Much of this ability stems from its acceptance of grammar-specific synthesis

parts which allow both permutation of the order of nonterminal children and the inclusion of terminal children. The effectiveness of this approach regarding translation and direct execution of the program tree should be researched to assess its potential, and further guidelines for grammar design should be developed.

This paper began with a discussion of modern programming environments and how syntax-directed editors fit into this scheme. As a final research suggestion, it is recommended that the SDE be placed into such an environment, accompanied by a set of viable grammars and interpretive tools that could act directly on the trees produced by the SDE. In this way the utility of the SDE as part of an overall programming environment may be assessed.

APPENDIX A

NOTEWORTHY SDE DATA TYPES

```
const alfalen = 10;
type alfarec = record
                 wrd: alfa;      {Berkeley's packed array
                                    [1..10] of char}
                 len: 0..alfalen;
               end;

  {Pointer and Record types used in the linked list of
   Grammar Rules: }

    grammar = ^grec;
    alist = ^arec;
    taggedalist = ^taggedarec;
    tntlist = ^tntrec;
    altlist = ^altrec;
    defnptr = ^defnrec;

    affixrec = record
                 a1, a2, a3: char;
               end;
    grec = record
             name: alfarec;
             next: grammar;
             case isalternation: boolean of
               true:  (alternatives: altlist);
               false: (defn: defnptr);
           end;
    altrec = record
               choice: char;
               choicedefn: defnptr;
               choicedisplay: alfa;
               next: altlist;
             end;
    defnrec = record
                anal: tntlist;
                syn: taggedalist;
                ntdict: alist;
              end;
    tntrec = record
               isnt: boolean;
               info: alfarec;
               affix: affixrec;
               next: tntlist;
             end;
    taggedarec = record
                   tag: alfarec;
                   avlist: alist;
                 end;
    arec = record
             attr: alfarec;
             val: alfarec;
             valisnt: boolean;
             affix: affixrec;
             next: alist;
             prev: alist;
           end;
```

```
{Types used in the Grammar's Sets: }

   setptr = ^setrec;
   setrec = record
              name: alfarec;
              members: set of char;
              next: setptr;
            end;

{Pointer and Record types used in the Program Tree: }

   nodeptr = ^prognode;
   childptr = ^childnode;

   prognode = record
                name: alfarec;
                syntax: defnptr;
                isnt: boolean;
                parent: nodeptr;
                childlist: childptr;
              end;
   childnode = record
                 path: alfarec;
                 child: nodeptr;
                 next: childptr;
               end;

{Record type that maintains the Current Position
 in the tree: }

   currloc = record
               cn: nodeptr;
               cp: childptr;
             end;

{Types used in the linked list of Legal Menu Choices: }

   choiceptr = ^choicerec;
   choicerec = record
                 choice: char;
                 descr: alfa;
                 next: choiceptr;
               end;
```

APPENDIX B
DESCRIPTION OF INPUT GRAMMARS TO THE SDE

The following paragraphs describe the format of a grammar suitable for input to the SDE. At the end of the appendix is the input grammar used to generate other grammars, which may be considered "programs" in the "language" of grammars. Because the listing is suitable for input to the SDE, it both demonstrates and restates the prose description below.

Certain semantic details should be noted at the outset. All strings described below must be no more than 10 characters in length. This is not checked by the grammar-generating grammar. A string of more than 10 characters will cause the SDE to print a warning message during initialization, and the SDE will truncate the string to 10 characters. Other special considerations involve the use of double quotes and double periods in place of single characters, as described below. Further semantic requirements are listed in Appendix C. Note also that, except where specified otherwise, items in a grammar are separated by a space.

An input grammar is a sequence of one or more rules followed by a sequence of zero or more sets. The sets are separated from the rules by a colon (":") on a separate line. The first rule must be a regular rule with a non-null synthesis part, as described below.

A rule is either a "regular" rule or an "alternation." A regular rule is composed of a name, an analysis part and a synthesis part. The name is a string (of up to 10 characters) and is followed by a period (".") to delimit it from the rest of the rule. The analysis and synthesis parts are each enclosed by a set of parentheses.

Alternation rules are composed of a name and one or more choices. The name is separated from the rest of the rule by a period followed by an "A" to distinguish the alternation from a regular rule. Further, the entire set of choices is enclosed in a set of parentheses. Each choice is composed of a single character (used to select the choice), analysis and synthesis parts, and a display string describing this choice in the menu. As above, the analysis and synthesis parts are each enclosed by their own set of parentheses. The choice character and display string are enclosed by quotes.

An analysis part is a sequence of terminals and nonterminals. Terminals are strings delimited by quotes. If a quote is to be part of a terminal, it is represented by two consecutive quotes, i.e., "". Nonterminals are names delimited by parentheses and followed by an affix as described below.

Synthesis parts consist of a node name followed by zero or more children. Each child is a path name and a childnode, which are separated from each other by a period. A childnode is either a terminal or nonterminal as described above. Note that the node name in a synthesis part is always followed by a space. Thus, even if the node has no children, a space will separate the name of the node from the closing parenthesis of the synthesis part. Note also that synthesis parts are entirely optional in any rule except the very first in the grammar. However, the opening and closing parentheses of the synthesis part must always be present.

An affix on a nonterminal in the analysis and synthesis parts of a rule may consist of up to three characters. The first character may be a "&", "+", "*", "?", "'", ".", or any of the 10 digits. If the character is a digit or "'" symbol, a second character is included which may be any of

the above symbols (excluding the digits and "'"). Also, if the first character (or second, if the first was a digit or "'") is a ".", a second (or third) character is added. This character may be any printable character at all.

A set consists of a name and a string of characters. The name is separated from the string by a period. Note that this string is not limited to 10 characters in length; in fact, it may extend over several lines. The string is ended by a double period. (A single period means the symbol "." is to be added to the string.)

The following is a grammar demonstrating the above syntax. It is also the grammar used by the SDE to create "programs" in this syntax, and is therefore self-describing as well as suitable for input tc the SDE:

```
grammar.((rule)+ "%%:" (set)* )
    (gram r.(rule)+ s.(set)* )

rule.A(
  "r"("%%" (name)& "." (anal)& "%\\(" (syn)? ")!!" )
      (reg n.(name)& a.(anal)& s.(syn)? ) "regular"
  "a"("%%" (name)& ".A(" (choice)+ ")" )
      (alt n.(name)& c.(choice)+ ) "altrnation")

name.((char)+ )
    (name c.(char)+ )

choice.("%\"" (char)& """" (anal)& "%\\(" (syn)? ") """
        (display)& """!!!" )
    (choice c.(char)& a.(anal)& s.(syn)? d.(display)& )

anal.("(" (tnt)+ ")" )
    (anal list.(tnt)+ )

tnt.A(
  "t"(""" (name)& """ " )
      (term n.(name)& ) "terminal"
  "n"("(" (name)& ")" (affix)& " ")
      (nterm n.(name)& a.(affix)& ) "nonterm")

syn.((node)& " " (child)* )
    (synpart node.(node)& c.(child)* )

node.((name)& )
    ()

child.((path)& "." (cnode)& )
    (child p.(path)& cn.(cnode)& )

path.((name)& )
    ()

cnode.((tnt)& )
     ()
```

```
affix.A(
  "ε"("ε" )
      (ε1 ) "single nt"
  "+"("+" )
      (+1 ) "Kleene +"
  "*"("*" )
      (*1 ) "Kleene *"
  "?"("?" )
      (?1 ) "optional"
  "'"("'" (affix2)ε )
      (prime 2.(affix2)ε ) "prime"
  "d"((digit)ε (affix2)ε )
      (prime2 d.(digit)ε 2.(affix2)ε ) "digit"
  "."("." (char)ε)
      (list1 c.(char)ε ) "list")

affix2.A(
  "ε"("ε" )
      (ε2 ) "single nt"
  "+"("+" )
      (+2 ) "Kleene +"
  "*"("*" )
      (*2 ) "Kleene *"
  "?"("?" )
      (?2 ) "optional"
  "."("." (char)ε )
      (list2 c.(char)ε ) "list")

display.((char)+ )
    (display c.(char)+ )

set.("%%" (name)ε "." (char)+ "..")
    (set n.(name)ε c.(char)+ )

:

char.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
  1234567890 ,.<>/?:;"'{}|@#$_-+=!\%..

digit.0123456789..
```

APPENDIX C

SEMANTIC RESTRICTIONS OF GRAMMAR DESIGN

The following rules apply to grammars to be used as input to the SDE. The syntax of such grammars may be found in Appendix B.

1) For a given regular rule or alternation choice:

   a) every nonterminal in the analysis part must also appear as a child in the synthesis part, and vice versa. The nonterminals must match identically, including their affixes. (The only exception to this rule is the "identity," which consists of exactly one nonterminal in the analysis part, and no synthesis part at all.) The order of the nonterminals in the analysis part is independent of their order in the synthesis part;

   b) the synthesis part may have additional terminal children, and the analysis part may have additional terminal strings -- these are not related to each other at all;

   c) the path name of each child for a given synthesis part must be unique;

   d) each nonterminal in the analysis part (and synthesis part) must be unique. If more than one appears, they are to be distinguished by their affixes: the first affix of all but one (or all of them, if preferred) will be a unique symbol selected from the set of digits and the prime ("'").

2) Across the entire grammar:

   a) every rule and set name must be unique;

   b) the tag name of every synthesis part must be unique;

c) every nonterminal name in analysis and synthesis parts must either be a rule name or a set name.

3) Every non-identity regular rule must have at least one nonterminal in its analysis and synthesis parts.

4) The first rule in the grammar must be a non-identity regular rule.

5) Individual choices of an alternation may:

a) be similar to non-identity regular rules in that their synthesis parts may include a tag and at least one nonterminal son;

b) have synthesis parts that have only a tag (and no children);

c) have synthesis parts with tags and terminal children only;

d) be identities that reference rules as described in a, b, or c above. (Note that the end rules would meet the "regular rule" syntax -- but those in b and c above must not be referenced as regular rules elsewhere in the grammar, since they violate "regular rule" semantics.) Note that identities may also reference other identities to form a "chain" leading to a rule as described in this section.

6) Choices of alternations, therefore, must NOT be sets, other alternations, or identities leading to sets or alternations.

APPENDIX D

INPUT GRAMMAR FOR EXAMPLE IN CHAPTER 3

```
sample.((A)& )
    (sample a.(A)& )

A.A(
  "1"((T)& )
      (a1 t.(T)& ) "A --> T"
  "2"((A)& " + " (T)& )
      (a2 opnd1.(A)& opnd2.(T)& oprtr."add" ) "A--> A + T")

T.A(
  "1"((F)& )
      (t1 f.(F)& ) "T --> F"
  "2"((T)& " * " (F)& )
      (t2 opnd1.(T)& cpnd2.(F)& oprtr."mpy" ) "T--> T * F")

F.A(
  "1"((char)& )
      (f1 c.(char)& ) "F --> char"
  "2"(" (" (A)& ")" )
      (f2 a.(A)& ) "F --> (A)")

:

char.qrs..
```

APPENDIX E

MINIGOL GRAMMAR

```
block.("%\begin" (decl)* (statement).; "%end!" )
    (block head.(decl)* body.(statement).; )

decl.("%\" (type)& (id)& "!" )
    (decl n.(id)& t.(type)& )

type.A(
  "n"("integer " )
      (int ) "integer"
  "r"("real " )
      (real ) "real")

statement.A(
  "a"((assign)& )
      () "assignment"
  "w"((whileloop)& )
      () "while loop"
  "b"((block)& )
      () "block"
  "i"((ifstat)& )
      () "if statmnt")

assign.("%\" (var)& ":= " (exp)& "!" )
    (asn d.(var)& s.(exp)& )

ifstat.("%\if " (relation)& " then" (statement)&
        (elsepart)? "!" )
    (if cond.(relation)& conseq.(statement)&
       alt.(elsepart)? )

elsepart.("%else" (statement)& )
    (else s.(statement)& )

whileloop.("%\while " (relation)& " do" (statement)& "!" )
    (while cond.(relation)& body.(statement)& )

relation.((exp)& (relop)& (exp)'& )
    (reln op.(relop)& 1.(exp)& 2.(exp)'& )

relop.A(
  "="(" = " )
      (eq ) "="
  "n"(" <> " )
      (ne ) "not ="
  "<"(" < " )
      (lt ) "<"
  ">"(" > " )
      (gt ) ">"
  "l"(" <= " )
      (le ) "<="
  "g"(" >= " )
      (ge ) ">=")

exp.A(
  "+"((exp)& " + " (term)& )
      (add 1.(exp)& 2.(term)& ) "exp + trm"
  "-"((exp)& " - " (term)& )
      (sub 1.(exp)& 2.(term)& ) "exp - trm"
  "*"((term)& " * " (factor)& )
```

```
        (mul1 1.(term)& 2.(factor)& ) "trm * fctr"
  "/"((term)& " / " (factor)& )
        (div1 1.(term)& 2.(factor)& )"trm / fctr"
  "("("(" (exp)& ")" )
        (exp1 e.(exp)& ) "(exp)"
  "#"((number)& )
        () "number"
  "v"((var)& )
        () "variable")

term.A(
  "*"((term)& " * " (factor)& )
        (mul2 1.(term)& 2.(factor)& ) "trm * fctr"
  "/"((term)& " / " (factor)& )
        (div2 1.(term)& 2.(factor)& ) "trm / fctr"
  "("("(" (exp)& ")" )
        (exp2 e.(exp)& ) "(exp)"
  "#"((number)& )
        () "number"
  "v"((var)& )
        () "variable")

factor.A(
  "("("(" (exp)& ")")
        (f e.(exp)& ) "(exp)"
  "#"((number)& )
        () "number"
  "v"((var)& )
        () "variable")

var.((id)& )
    ()

id.((char)+ )
    (id n.(char)+ )

number.((digit)+ )
    (num val.(digit)+ )

:

char.abcdefghijklmnopqrstuvwxyz_..

digit.0123456789..
```

APPENDIX F
DESCRIPTION OF THE "TERM" FILE

The "TERM" file required by the SDE during initialization is a text file consisting of 5 lines of integers. The first four lines in the file contain information about a particular terminal's commands to activate and inactivate inverse video, clear the screen and move the cursor to the beginning of the SDE menu display on the screen. The fifth line contains dimensional information about the display area.

The information in the first four lines follows the same pattern. The first integer on each line represents the number of integers that follow on that line. (This tells the SDE how many integers are to be read on the line. Since the SDE reads the information as integers, it can not detect ends of lines without this knowledge.) The second integer tells the SDE how many spaces on the screen will be physically occupied by the rest of the sequence on that line. The remaining integers represent a sequence of characters that will cause the terminal to perform as desired. Conversion of the integers to characters is done using the Pascal "chr" function; when using the ASCII character set, therefore, the number "65" represents the character "A", while the number "17" represents the character "control-W".

The purpose of the second integer on each of the first four lines is to inform the unparser how many columns to count when invoking the sequence. Usually, activating inverse video causes no spaces to be used on the screen for those terminals that support it. However, for terminals that do not support inverse video, a printable sequence of characters must be used instead. These characters will take

up space on the display, and the SDE must account for these positions to determine where to force carriage returns on long lines.

Note that the SDE uses these integer strings as "black boxes" to achieve desired effects. This not only achieves terminal independence, but it also allows the user some freedom in customizing his display. For example, Figure 4.1 showed one means of displaying the CP on terminals without inverse video. The user could easily have used "<<<" and ">>>" instead of "-}" and "{-" by modifying the first and second lines of the "TERM" file. Note, however, that it is a good idea to include at least a single space (ASCII number 32) in the "inverse off" sequence, especially on those terminals that do support inverse video. This is because the CP may reference a "nil" node, which has no display of its own. Activating and inactivating inverse video in succession (without printing a space in between these actions) will have no visible effect at all on most terminals, and the CP will seem to disappear.

The fourth line of integers moves the cursor to the beginning of the menu display on the screen. On a 24-line screen, this should be the eighteenth line, first column. This position affords a sufficient amount of space for the menu while leaving most of the screen for the program display. However, in grammars with very long alternation lists, or on a screen where only three commands will fit on a single line, the menu must begin higher on the screen to prevent scrolling. Customization of the menu for the terminal and particular grammar in use is a refining process.

The final line of the "TERM" file provides information on the dimensions of the screen. Like the other lines, the first integer tells the SDE how many integers follow; however, the line contains no integer describing how many

spaces are used by the rest of the line. Rather, the second integer represents the number of lines on the display (usually 24 or 16); the third integer contains the number of columns (usually 80 or 64); and the fourth integer states the line on which the menu commences. Note that this last integer is required because this information is not always obvious from the sequence on the fourth line -- each terminal has its own algorithm for moving the cursor.

The following figure represents the "TERM" file for the VT100 terminal. Brackets have been added to include comments, which would not appear at all in the actual "TERM" file.

```
5  0  27 91 55  109  {sequence to activate inverse video}
6  1  32 27 91  48 109 {turns inverse video off.  Note
                        the "1" and the "32" for a space}
8  0  27 91 59  72 27 91 74  {clears the screen}
11 0  27 91 49  57 59 49 72 27 91 74 {moves cursor to
                                     {18th line, 1st column}
3  24 80 18  {screen data}
```

Figure F.1 'Term' File for VT100 Terminal

APPENDIX G

SAMPLE GRAMMAR FOR A DATABASE APPLICATION

```
db.((course)+ )
    (db c.(course)+ )

course.("%Course#: " (crsnum)& "   Title: " (title)&
        "   Descr: " (letter)+ "%\Prereqs:" "\" (prereq)*
        "!%" "Offerings:" "\" (offerings)* "!!%" )
    (course 1.(crsnum)& 2.(title)& 5.(letter)+ 3.(prereq)*
      4.(offerings)* )

prereq.("%" "Course#: " (crsnum)& "   Title: " (title)& )
    (prereq 1.(crsnum)& 2.(title)& )

crsnum.((letter)& (digit)& (digit)'& )
    (crsnum 1.(letter)& 2.(digit)& 3.(digit)'& )

title.((letter)+ )
    (title 1.(letter)+ )

offerings.("%Date: " (digit)+ "   " "Location: "
           (letter)+ "   Format:" " " (letter)& (digit)&
           "%\Teachers" ":\" (teacher)* "!%Students" ":\"
           (student)* "!!" )
    (offerings 1.(digit)+ 2.(letter)+ 3.(letter)& 4.(digit)&
      5.(teacher)* 6.(student)* )

teacher.("%" (prsndat)& )
    (teacher 1.(prsndat)& )

prsndat.("Emp#: " (digit)+ "   Name: " (letter)+ )
    (prsndat 1.(digit)+ 2.(letter)+ )

student.("%" (prsndat)& "   Grade: " (letter)& )
    (student 1.(prsndat)& 2.(letter)& )

:

letter.ABCDEFGHIJKLMNOPQRSTUVWXYZ., ..

digit.1234567890..
```

APPENDIX H

ALTERNATE GRAMMAR FOR GENERATING GRAMMARS

The following grammar represents Argot, a grammar-describing format. It is suitable for input to the SDE and subsequent use as a grammar-generating grammar by users who prefer its format over the syntax of the grammar in Appendix B.

```
grammar.((rule)+ "%%sets:" (set)* )
    (gram r.(rule)+ s.(set)* )

rule.A(
  "r"("%%" (name)& ": " (anal)& "% => " (syn)? "." )
      (reg n.(name)& a.(anal)& s.(syn)? ) "regular"
  "a"("%%" (name)& ": {" (choice).; "}" )
      (alt n.(name)& c.(choice).; ) "altrnation")

name.((char)+ )
    (name c.(char)+ )

choice.("%\" (char)& "(" (display)& "): " (anal)& "%\\=> "
        (syn)? "!!!" )
    (choice c.(char)& a.(anal)& s.(syn)? d.(display)& )

anal.((tnt)+ )
    (anal list.(tnt)+ )

tnt.A(
  "t"("""" (name)& """ " )
      (term n.(name)& ) "terminal"
  "n"("<" (name)& ">" (affix)& " " )
      (nterm n.(name)& a.(affix)& ) "nonterm")

syn.((node)& ": " (child)* )
    (synpart node.(node)& c.(child)* )

node.((name)& )
    ()

child.((path)& "=" (cnode)& )
    (child p.(path)& cn.(cnode)& )

path.((name)& )
    ()

cnode.((tnt)& )
    ()

affix.A(
  "&"("" )
      (&1 ) "no affix"
  "+"("+" )
      (+1 ) "+"
  "*"("*" )
      (*1 ) "*"
  "?"("?" )
```

```
      (?1 ) "?"
  "'"("'" (affix2)& )
      (prime 2.(affix2)& ) "prime"
  "."((char)& "..." )
      (list c.(char)& ) "list")

affix2.A(
  "&"(" " )
      (&2 ) "no affix"
  "+"("+" )
      (+2 ) "+"
  "*"("*" )
      (*2 ) "*"
  "?"("?" )
      (?2 ) "?"
  "."((char)& "..." )
      (list c.(char)& ) "list")

display.((char)+ )
    (dsply c.(char)+ )

set.("%%" (name)& " = {" (char)., "}" )
    (set n.(name)& c.(char)., )

:

char.ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
  1234567890 ,.<>/?:;"'{}|@#$_-+=!\%..
```

As stated, the above grammar is suitable for designing grammars in a different format from that of Appendix B. An example of the format described above is the following, which is the same grammar in the new format.

```
grammar: <rule>+ "%%sets:" <set>*
 => gram: r=<rule>+ s=<set>* .

rule: {
  r(regular): "%%" <name>  ": " <anal>  "% => " <syn>? "."
      => reg: n=<name>  a=<anal>  s=<syn>? ;
  a(altrnation): "%%" <name>  ": {" <choice>;... "}"
      => alt: n=<name>  c=<choice>;... }

name: <char>+
 => name: c=<char>+ .

choice: "%\" <char>  "(" <display>  "): " <anal>  "%\\=> "
  <syn>? "!!!"
 => choice: c=<char>  a=<anal>  s=<syn>? d=<display>  .

anal: <tnt>+
 => anal: list=<tnt>+ .

tnt: {
  t(terminal): """" <name>  """ "
      => term: n=<name>  ;
  n(nonterm): "<" <name>  ">" <affix>  " "
      => nterm: n=<name>  a=<affix>  }

syn: <node>  ": " <child>*
 => synpart: node=<node>  c=<child>* .

node: <name>
 => .
```

```
child: <path>  "=" <cnode>
 => child: p=<path>  cn=<cnode>  .

path: <name>
 => .

cnode: <tnt>
 => .

affix: {
  &(no affix): " "
       => &1: ;
  +(+): "+"
       => +1: ;
  *(*): "*"
       => *1: ;
  ?(?): "?"
       => ?1: ;
  '(prime): "'" <affix2>
       => prime: 2=<affix2>  ;
  .(list): <char>  "..."
       => list: c=<char>  }

affix2: {
  &(no affix): " "
       => &2: ;
  +(+): "+"
       => +2: ;
  *(*): "*"
       => *2: ;
  ?(?): "?"
       => ?2: ;
  .(list): <char>  "..."
       => list: c=<char>  }

display: <char>+
 => dsply: c=<char>+ .

set: "%" <name>  " = {" <char>,... "}"
 => set: n=<name>  c=<char>,... .

sets:

char = {ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
  1234567890 ,.<>/?:;"'{}|@#$_-+=!\%}
```

APPENDIX I

MINIGOL GRAMMAR MODIFIED FOR PASCAL COMPATIBILITY

The following is a modification of the "Minigol" grammar presented in Appendix E. The purpose of the modification is to make minigol compatible with the Pascal grammar in Appendix J. See Section 5.C for the implications of this compatibility.

```
block.("%\begin" (decl)? (stateblock)& "%end!" )
    (block head.(decl)? body.(stateblock)& )

stateblock.((statement).; )
    (stateblock s.(statement).; )

decl.((decl2)+ )
    (decl d.(decl2)+ )

decl2.("%\" (type)& (id)& "!" )
    (decl2 n.(id)& t.(type)& )

type.A(
  "n"("integer " )
      (int ) "integer"
  "r"("real " )
      (real ) "real")

statement.A(
  "a"((assign)& )
      () "assignment"
  "w"((whileloop)& )
      () "while loop"
  "b"((block2)& )
      () "block"
  "i"((ifstat)& )
      () "if statmnt")

block2.("%\begin" (decl)* (stateblock)& "%end!" )
    (block2 head.(decl)* body.(stateblock)& )

assign.("%\" (var)& ":= " (exp)& "!" )
    (asn d.(var)& s.(exp)& )

ifstat.("%\if " (relation)& " then" (statement)&
        (elsepart)? "!" )
    (if cond.(relation)& conseq.(statement)&
      alt.(elsepart)? )

elsepart.("%else" (statement)& )
    (else s.(statement)& )

whileloop.("%\while " (relation)& " do" (statement)& "!" )
    (while cond.(relation)& body.(statement)& )

relation.((exp)& (relop)& (exp)'& )
    (rel op.(relop)& 1.(exp)& 2.(exp)'& )
```

```
relop.A(
  "="("="  )
       (eq ) "="
  "n"("<>")
       (ne ) "not ="
  "<"("< ")
       (lt ) "<"
  ">"(" >")
       (gt ) ">"
  "l"("<=")
       (le ) "<="
  "g"(">=")
       (ge ) ">=")

exp.A(
  "+"((exp)& " + " (term)& )
       (add 1.(exp)& 2.(term)& ) "exp + trm"
  "-"((exp)& " - " (term)& )
       (sub 1.(exp)& 2.(term)& ) "exp - trm"
  "*"((term)& " * " (factor)& )
       (mul1 1.(term)& 2.(factor)& ) "trm * fctr"
  "/"((term)& " / " (factor)& )
       (div1 1.(term)& 2.(factor)& ) "trm / fctr"
  "("("(" (exp)& ")" )
       (exp1 e.(exp)& ) "(exp)"
  "#"((number)& )
       () "number"
  "v"((var)& )
       () "variable")

term.A(
  "*"((term)& " * " (factor)& )
       (mul2 1.(term)& 2.(factor)& ) "trm * fctr"
  "/"((term)& " / " (factor)& )
       (div2 1.(term)& 2.(factor)& ) "trm / fctr"
  "("("(" (exp)& ")" )
       (exp2 e.(exp)& ) "(exp)"
  "#"((number)& )
       () "number"
  "v"((var)& )
       () "variable")

factor.A(
  "("("(" (exp)& ")" )
       (f e.(exp)& ) "(exp)"
  "#"((number)& )
       () "number"
  "v"((var)& )
       () "variable")

var.((id)& )
    ()

id.( (char)+ )
    (id n.(char)+ )

number.((digit)+ )
    (num val.(digit)+ )

:

char.abcdefghijklmnopqrstuvwxyz_..

digit.0123456789..
```

APPENDIX J

PASCAL SUBSET GRAMMAR

```
block.("%\" (decl)? (stateblock)& ";" )
    (block d.(decl)? body.(stateblock)& )

stateblock.("%\begin" (statement).; "%end!" )
    (stateblock body.(statement).; )

decl.("%\var\" (decl2)+ "!!" )
    (decl d.(decl2)+ )

decl2.("%" (id)& ": " (type)& ";" )
    (decl2 n.(id)& t.(type)& )

type.A(
  "n"("integer" )
      (int ) "integer"
  "r"("real" )
      (real ) "real")

statement.A(
  "a"((assign)& )
      () "assignment"
  "w"((whileloop)& )
      () "while loop"
  "b"((block2)& )
      () "block"
  "i"((ifstat)& )
      () "if statmnt")

block2.((stateblock)& )
    (block2 x."nil" body.(stateblock)& )

assign.("%\" (var)& ":= " (exp)& "!" )
    (asn d.(var)& s.(exp)& )

ifstat.("%\if " (relation)& " then" (statement)&
        (elsepart)? "!" )
    (if cond.(relation)& consq.(statement)&
      alt.(elsepart)? )

elsepart.("%else" (statement)& )
    (else s.(statement)& )

whileloop.("%\while " (relation)& " do" (statement)& "!" )
    (while cond.(relation)& body.(statement)& )

relation.((exp)& (relop)& (exp)'& )
    (rel op.(relop)& 1.(exp)& 2.(exp)'& )

relop.A(
  "="("= " )
      (eq ) "="
  "n"("<> ")
      (ne ) "not ="
  "<"("< ")
      (lt ) "<"
  ">"("> ")
      (gt ) ">"
  "l"("<= ")
      (le ) "<="
```

```
  "g"(" >= ")
      (ge ) ">=")

exp.A(
  "+"((exp)& " + " (term)& )
      (add 1.(exp)& 2.(term)& ) "exp + trm"
  "-"((exp)& " - " (term)& )
      (sub 1.(exp)& 2.(term)& ) "exp - trm"
  "*"((term)& " * " (factor)& )
      (mul1 1.(term)& 2.(factor)& ) "trm * fctr"
  "/"((term)& " / " (factor)& )
      (div1 1.(term)& 2.(factor)& ) "trm / fctr"
  "("("(" (exp)& ")" )
      (exp1 e.(exp)& ) "(exp)"
  "#"((number)& )
      () "number"
  "v"((var)& )
      () "variable")

term.A(
  "*"((term)& " * " (factor)& )
      (mul2 1.(term)& 2.(factor)& ) "trm * fctr"
  "/"((term)& " / " (factor)& )
      (div2 1.(term)& 2.(factor)& ) "trm / fctr"
  "("("(" (exp)& ")" )
      (exp2 e.(exp)& ) "(exp)"
  "#"((number)& )
      () "number"
  "v"((var)& )
      () "variable")

factor.A(
  "("("(" (exp)& ")" )
      (f e.(exp)& ) "(exp)"
  "#"((number)& )
      () "number"
  "v"((var)& )
      () "variable")

var.((id)& )
    ()

id.((char)+ )
    (id n.(char)+ )

number.((digit)+ )
    (num val.(digit)+ )

:

char.abcdefghijklmnopqrstuvwxyz_..

digit.0123456789..
```

LIST OF REFERENCES

1. Katzan, H., Computer Systems Organization and Programming, Science Research Associates, Inc., 1976.

2. Rule, W. P., Finkenaur, R. G., and Patrick, F. G., FORTRAN IV Programming, Prindle, Weber and Schmidt, Inc., 1973.

3. Sandewall, E., "Programming in an Interactive Environment: the LISP Experience," ACM Computing Surveys, v. 10:1, March 1978.

4. Winograd, T., "Breaking the Complexity Barrier (Again)," Proceedings of the ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages: Information Retrieval, November 1973.

5. Teitelman, W., and Masinster, L., "The Interlisp Programming Environment," Computer, v. 14:4, April 1981.

6. Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," Communications of the ACM, v. 24:9, September 1981.

7. Barstow, D. R., "A Lisplay-Oriented Editor for Interlisp," in Interactive Programming Environments, edited by Barstow, D. R., Shrobe, H. E., and Sandewall, E., McGraw-Hill Book Company, 1984.

8. Bottos, B. A., and Kintala, C. M. R., "Generation of Syntax-Directed Editors with Text-Oriented Features," Bell System Technical Journal, v. 62:10, December 1983.

9. Wood, S. R., "Z -- the 95% Program Editor," Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, v. 16:6, June 1981.

10. Stromfors, D., and Jonesjo, L., "The Implementation and Experiences of a Structure-Oriented Text Editor," Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, v. 16:6, June 1981.

11. Walker, J., "The Document Editor: A Support Environment for Preparing Technical Documents," Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, v. 16:6, June 1981.

12. Allen, T., Nix, R., and Perlis, A., "PEN: A Hierarchical Document Editor," Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, v. 16:6, June 1981.

13. Donzeau-Gouge, V., and others, "Programming Experiences Based on Structured Editors: the MENTOR Experience," in Interactive Programming Environments, edited by Barstow, D. R., Shrobe, H. E., and Sandewall, E., McGraw-Hill Book Company, 1984.

14. Fraser, C. W., "Syntax-Directed Editing of General Data Structures," Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, v. 16:6, June 1981.

15. Naval Postgraduate School Report NPS52-81-014, The Automatic Generation of Syntax Directed Editors, by B. J. MacLennan, October 1981.

16. Shockley, W. R., and Haddow, D. P., A Conceptual Framework for Grammar-Driven Synthesis, M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1980.

17. Aho, A. V., and Ullman, J. D., Principles of Compiler Design, Addison-Wesley Publishing Company, 1977.

18. Barrett, W. A., and Couch, J. D., Compiler Construction: Theory and Practice, Science Research Associates, Inc., 1979.

19. Hopcroft, J. E., and Ullman, J. D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Publishing Company, 1979.

20. MacLennan, B. J., Semantic and Syntactic Specification and Extension of Languages, Ph.D. Thesis, Purdue University, West Lafayette, Indiana, 1975.

21. MacLennan, B. J., Principles of Programming Languages: Design, Evaluation, and Implementation, Holt, Rinehart, and Winston, 1983.

22. Landin, P. J., "The Mechanical Evaluation of Expressions," Computer Journal, v. 6:4, January 1964.

23. MacLennan, B. J., The Four Forms of Omega, forthcoming technical report, Naval Postgraduate School, Monterey, California.

24. Date, C. J., An Introduction to Database Systems, 3d ed., Addison-Wesley Publishing Company, 1982.

INITIAL DISTRIBUTION LIST

No.		Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
4.	Professor Bruce J. MacLennan, Code 52ml Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
5.	Associate Professor Daniel L. Davis, Code 52vv Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
6.	Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943	1
7.	Captain George M. Tilley, Jr. Department of Geography and Computer Science United States Military Academy West Point, New York 10996	4

213170

Thesis
T482 Tilley
c.1 The design, implementation and application of a table-driven, syntax-directed editor.